PS4

February 27, 2019

1 INFO371 Problem set 4: k-NN, TF-IDF

1.0.1 Yasmine Hejazi

Deadline: Tue, Feb 25th midnight
Collaborator: Kishore Vasan

1.1 Introduction

This is the firstrst homework where you have to implement yourself some of the common ML algorithms. In particular, you will program k-NN with cosine similarity, using both bag-of-words and TF-IDF approach.

Before you start, ensure you have read:

- Daume "A course in machine learning", Ch 3.1 - 3.3. This a good introduction to k-NN.
- my lecture notes, titled machineLearning, the chapter about k-NN. This includes metric distance, cosine distance, and TF-IDF. Feel free to give some feedback so I can improve on what is unclear. (The currect version is a quick hack.)

You will use two types of data: first, various texts from Cantenbury corpus with several books added from Project Gutenberg, and thereafter Rotten Tomatoes, brief movie reviews. In case of the texts, your task is to find the correct source of the text, in case of tomatoes, to predict if the review is for a "rotten" or "fresh" movie.

There is also a separate file with some example python code where you can find chunks of useful code for this problemset.

Please submit a) your code (notebooks, rmd, whatever) and b) the lab in a final output form (html or pdf). Unlike PS1, you are free to choose either R or python for solving this problem set. While some of the intermediate output may be informative, please don't include too many lines of it in your solutions!

Note: it includes questions you may want to answer on paper instead of computer. You are welcome to do it but please include the result as an image into your final file.

1.2 1 Where are these texts coming from?

The data file texts.csv contains the texts you have to classify. It contains the following variables:

- **name**: name of the original file. It is usually author-name form and it should be fairly easy to find the original text in most cases.

- **size**: size of the original text, in bytes
- **lines**: size of the original text, in lines
- **chunkid**: chunk id, from 1 and growing, see chunk. If you want to re-assemble the original texts, you just have to put these next to each other in the order of chunkid.
- **chunk**: a page of text. It is not really a page, just 25 lines of text, whatever happened to be on those 25 lines.

Text chunks are just verbatim texts that may contain all kind of characters, including newlines. Note the file is tab separated and uses quotes for strings.

Your task is to read all the texts, convert these to a) bag-of-words, and b) TF-IDF-s, and predict the correct source using k-NN.

1.2.1 1.1 Bag of words

First, let's use bag-of-words (BOW) approach.

1. Load the data. Print out a few lines of it to inspect it's structure.
2. Inspect some of the texts. Note that chunkid 1 corresponds to the first page of the text.
3. List all the text sources listed in variable name. How many different texts does the data contain?

```
In [1]: #!/usr/bin/env python3
        import pandas as pd
        import numpy as np
        from sklearn.feature_extraction.text import CountVectorizer
```

```
In [2]: texts = pd.read_csv("texts.csv", sep='\t')
        # note: as the texts may contain various symbols, including tabs and commas,
        # you may to have specify 'sep' and maybe other extra arguments

        texts.head()
```

```
Out[2]:                              name    size  lines  chunkid  \
        0  balbulus-early-life-charlemagne  259062   4394        1
        1  balbulus-early-life-charlemagne  259062   4394        2
        2  balbulus-early-life-charlemagne  259062   4394        3
        3  balbulus-early-life-charlemagne  259062   4394        4
        4  balbulus-early-life-charlemagne  259062   4394        5

                                                   chunk
        0  \nTitle: Early Lives of Charlemagne by Eginhar...
        1  \n\nThe notes, keyed to line numbers in the so...
        2  \n         From a bronze statuette in the Musé...
        3  \n                 _A lui finit la dissolution ...
        4  public opinion in regard to the meaning of fal...
```

```
In [3]: print("Number of texts: %i" % texts.shape[0])
        print("Number of titles: %i" % len(np.unique(texts.name.astype('str'))))
```

```
Number of texts: 12924
Number of titles: 29
```

```
In [4]: first_pages = texts[texts['chunkid'] == 1]
        first_pages

        # inspect alice in wonderland
        print(first_pages[first_pages['name'] == 'carroll-alice-wonderland'].chunk.values[0])
```

```
                ALICE'S ADVENTURES IN WONDERLAND

                           Lewis Carroll

                THE MILLENNIUM FULCRUM EDITION 2.9



                                CHAPTER I

                            Down the Rabbit-Hole


  Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do:  once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
thought Alice `without pictures or conversation?'

  So she was considering in her own mind (as well as she could,
for the hot day made her feel very sleepy and stupid), whether
the pleasure of making a daisy-chain would be worth the trouble
```

```
In [5]: print("All text sources: ", np.unique(texts.name), "\n")
        print("Number of text sources: ", len(np.unique(texts.name)))
```

```
All text sources:  ['balbulus-early-life-charlemagne' 'beesly-queen-elizabeth' 'bible'
 'carroll-alice-wonderland' 'chipman-earliest-electromagnetic-instruments'
 'cia-world-factbook-1992' 'eckstein-quintus-claudius'
 'fisher-quaker-colonies' 'gallienne-quest-of-golden-girl'
 'gordon-quiet-talks-crowned-christ' 'hardy-madding-crowd'
 'infiltrating-open-systems' 'kant-metaphysical-elements-ethics'
 'karn-snowflakes' 'milton-paradise-lost'
 'naval-academy-sound-military-decision' 'newsgroup'
 'paper-compact-hash-tables' 'paper-data-compression'
 'paper-logical-implementation-of-arithmetic'
 'paper-programming-by-example' 'paper-search-for-autonomy'
```

```
 'selected-polish-tales' 'shakespeare-as-you-like-it'
 'unamuno-tragic-sense-of-life' 'vaneeden-quest'
 'webster-early-european-history' 'why-speech-output'
 'workshop-proceedings']

Number of text sources:  29
```

When you feel you know the data well enough, it's time to split it into testing and validating parts.

Warning: start slow. The dataset contains 13,000 texts and 65,000 unique words. If you load in everything, the BOW matrix takes approximately 8GB of RAM. I recommend to start with very small datasets, say 10 pages for both training and validation. When your code works, start increasing the size. Stop where it gets too slow, you don't have to run everything (it takes 18G RAM). But note that if you drastically scale down the amount of data, you should also select only a few sources. Otherwise you may have no examples from certain sources to compare to.

Now when you have created your training and testing sets, it is time to create the dictionary. See the example code for python examples, there are many examples on the web about how to do it with R (see text mining with R or text2vec package).

This process contains two steps:

1) create a dictionary of all your texts.

2) recode all the texts as BOW vector. See the lecture notes for explanation about BOW.

Note you may want to feed in all your data, i.e. both training and validation data into the dictionary. This ensures all words in your validation are represented in the dictionary

```
In [69]: from sklearn.model_selection import train_test_split


         ntrain = 40
         nVal = 10

         train = texts.sample(n=ntrain,random_state = 1)
         val = texts.loc[~texts.index.isin(train_df)].sample(n=nVal,random_state = 1)

         # split intro train/val
         X_train = train.chunk
         X_val = val.chunk
         y_train = train.name
         y_val = val.name

In [70]: # for simplicity, I recommend to create dictionary based on the
         # merged training and validation data
         sentences = np.concatenate((X_train.values, X_val.values),axis=0)

         ## initialize the vectorizer
         vectorizer = CountVectorizer(min_df=0)
```

```
        ## create the dictionary
        vectorizer.fit(sentences)
        # `fit` builds the vocabulary
        ## transform your data into the BOW array
        X = vectorizer.transform(sentences).toarray()

        print(X, '\n')
        print('Number of rows:', len(X))
```

```
[[0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 ...,
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]]

Number of rows: 50
```

Now you have all your (training and validation) texts in BOW form. This is great, we just transformed texts into (numeric) vectors! Now implement cosine similarity between these vectors.

Ensure you have read the lecture notes about cosine similarity. Write a function that takes in two vectors, x and y, and returns the corresponding cosine similarity c(x,y). Test if c(x, x) = 1, test also a few other vectors.

```
In [71]: def cos_sim(a, b):
             """Takes 2 vectors a, b and returns the cosine similarity according to the defini
             dot_product = np.dot(a, b)
             norm_a = np.linalg.norm(a)
             norm_b = np.linalg.norm(b)
             return dot_product / (norm_a * norm_b)

         cos_sim(X[1], X[1])
```

```
Out[71]: 1.0000000000000002
```

Finally, implement k-NN. I mean implement it yourself, don't use pre-existing libraries. I recommend to do 1-NN first, and when this works to extend it to k-NN. You need an algorithm along these lines:

(a) Pick a validation case y (later you will loop over all of these).

(b) for each vector in the training set xi, compute the cosine similarity c(y, xi). Store this number, and ensure you know which xi corresponds to each c value.

(c) Now order all the cosine similarities you just computed in an increasing order.

(d) Pick the k highest c-s. These correspond to your k nearest neighbors! Ensure you know which texts these are.

Now you know your nearest neighbors. Organize a majority voting among them so you will learn which text is the most popular among them. I recommend to create a frequency table and pick the most common text source name based on this.

Compute accuracy (percentage of correct predictions). How good is your algorithm?

```
In [72]: y = X[-1]
         train_cases = X[:40]
         result = []
         correct = []

         for i,j in enumerate(train_cases):
             result.append((i, cos_sim(y, j)))

         ordered_result = sorted(result, key=lambda tup: tup[1], reverse=True)

         print("Predicted:", train.iloc[ordered_result[0][0], 0])
         print("Correct:", val.iloc[9, 0])
```

```
Predicted: webster-early-european-history
Correct: webster-early-european-history
```

```
In [73]: # get accuracy of predicted responses
         def accuracy(responses):
             return float(sum(i == 1 for i in responses)) / float(len(responses))
```

```
In [80]: ntrain = 1000
         nVal = 100

         train = texts.sample(n=ntrain,random_state = 123)
         val = texts.loc[~texts.index.isin(train)].sample(n=nVal,random_state = 123)

         X_train = train.chunk
         X_val = val.chunk
         y_train = train.name
         y_val = val.name

         sentences = np.concatenate((X_train.values, X_val.values),axis=0)

         vectorizer = CountVectorizer(min_df=0)
         vectorizer.fit(sentences)
         X = vectorizer.transform(sentences).toarray()
```

```
In [81]: train_cases = X[:ntrain]
         correct = []

         for i in range(ntrain, ntrain + nVal):
             val_case = X[i]
             result = []
```

```
            for j,k in enumerate(train_cases):
                result.append((j, cos_sim(k, val_case)))

            ordered_result = sorted(result, key=lambda tup: tup[1], reverse=True)
            # add 1 if prediction is correct
            if train.iloc[ordered_result[0][0], 0] == val.iloc[i-1000, 0]:
                correct.append(1)
            else:
                correct.append(0)


        print("Accuracy:", accuracy(correct))
```

```
Accuracy: 1.0
```

Now that we've tried this, let's look into other values for k!

```
In [87]: def knn(k):
             print("k:", k)
             correct = []

             for i in range(ntrain, ntrain + nVal):
                 val_case = X[i]
                 result = []

                 for j, m in enumerate(train_cases):
                     result.append((j, cos_sim(m, val_case)))

                 ordered_result = sorted(result, key=lambda tup: tup[1],reverse=True)[:k]
                 vals = [y_train.iloc[o[0]] for o in ordered_result]
                 most_common = max(map(lambda val: (vals.count(val), val), set(vals)))[1]

                 if str(most_common) == y_val.iloc[i-ntrain]:
                     correct.append(1)
                 else:
                     correct.append(0)
             print("Accuracy:", accuracy(correct))
```

```
In [ ]: Load the data. Inspect it a little bit to see how it is coded and organized. How many
```

```
In [89]: import time
         for l in [1,5,25]:
             start = time.time()
             knn(l)
             end = time.time()
             print("Time elapsed:",end-start)
```

```
k: 1
Accuracy: 1.0
Time elapsed: 9.062533855438232
k: 5
Accuracy: 0.64
Time elapsed: 9.342237949371338
k: 25
Accuracy: 0.44
Time elapsed: 9.526848077774048
```

Our best model is when **k = 1** with an accuracy of 1.0. Accuracy decreases as we increase k.

1.2.2 1.2 TF-IDF

BOW is great but could be even greater. Can we get a better result using TF-IDF? TF-IDF is simply a way how to weigh the word frequency in a more informative way.

1. Implement TF-IDF transformation. Note: implement it yourself, don't rely on existing libraries! This involves manipulating your training and validation data matrices, nothing else needs to be done. Example will be given in the notes.
2. Now ensure that your cosine similarity you implemented earlier also works for vectors in TF-IDF form. It should, without any modications, if you implemented in well.
3. Run your k-NN with cosine similarity algorithm again, using several k values, like 1,5,25. The algorithm should not need any modications.
4. Finally, comment your results. How accurate is BOW versus TF-IDF? How does choice of k change the results? Is BOW or TF-IDF faster to run?

```
In [91]: def knn_tfidf(k):
             print("k:", k)
             correct = []
             for i in range(ntrain, ntrain + nVal):
                 val_case = tfidf[i]
                 result = []

                 for j, m in enumerate(train_tfidf):
                     result.append((j, cos_sim(m, val_case)))

                 ordered_result = sorted(result, key=lambda tup: tup[1], reverse=True)[:k]
                 vals = [y_train.iloc[o[0]] for o in ordered_result]
                 most_common = max(map(lambda val: (vals.count(val), val), set(vals)))[1]

                 if str(most_common) == y_val.iloc[i-ntrain]:
                     correct.append(1)
                 else:
                     correct.append(0)

             print("Accuracy:", accuracy(correct))
```

```
In [92]: tf = X.T #term frequency

         idf = np.log(tf.shape[1] / ( 1 + sum(tf != 0) ))
         idf = np.diag(idf)

         tfidf = np.dot(tf,idf).T

         print(tf.shape)
         print(idf.shape)
         print(tfidf.shape, '\n')

         # normalize
         tfidf = tfidf/np.sqrt(np.sum(tfidf**22))

         # train datasets
         train_tfidf = tfidf[:ntrain]

         for l in [1, 5, 25]:
             start = time.time()
             knn_tfidf(l)
             end = time.time()
             print("Time elapsed:", end-start)
```

```
(19029, 1100)
(1100, 1100)
(1100, 19029)
k: 1
Accuracy: 1.0
Time elapsed: 84.35733389854431
k: 5
Accuracy: 0.64
Time elapsed: 79.55618381500244
k: 25
Accuracy: 0.44
Time elapsed: 85.2572090625763
```

How accurate is BOW versus TF-IDF? How does choice of k change the results? Is BOW or TF-IDF faster to run?

Accuracy appears to be the same for both models when k = 1, 5, and 25. The change in the choice of k remains to have the same pattern, where accuracy decreases as k increases. BOW is much faster to run than TD-IDF.

1.3 2. Are tomatoes fresh or rotten?

Our next task is to use your freshly minted methods for classifying the Rotten Tomatoes movie reviews. Please familiarize yourself a little bit with the webpage. Briey, approved critics can write reviews for movies, and evaluate the movie as fresh or rotten. The webpage normally shows a

short quote from each critic, and whether it was evaluated as fresh or rotten. You will work on these quotes below.

Load the data. Inspect it a little bit to see how it is coded and organized. How many cases do you have?

```
In [97]: rotten = pd.read_csv("reviews.csv")
         print("Number of cases:", len(rotten))
         rotten.head()
```

```
Number of cases: 13442
```

Out[97]:

	critic	fresh	imdb	link	publication	quote	review_date	rtid	title
0	Derek Adams	fresh	114709	http://www.timeout.com/film/reviews/87745/toy-...	Time Out	So ingenious in concept, design and execution ...	2009-10-04 00:00:00	9559	Toy Story
1	Richard Corliss	fresh	114709	http://www.time.com/time/magazine/article/0,91...	TIME Magazine	The year's most inventive comedy.	2008-08-31 00:00:00	9559	Toy Story
2	David Ansen	fresh	114709	http://www.newsweek.com/id/104199	Newsweek	A winning animated feature that has something ...	2008-08-18 00:00:00	9559	Toy Story
3	Leonard Klady	fresh	114709	http://www.variety.com/review/VE1117941294.htm...	Variety	The film sports a provocative and appealing st...	2008-06-09 00:00:00	9559	Toy Story
4	Jonathan Rosenbaum	fresh	114709	http://onfilm.chicagoreader.com/movies/capsule...	Chicago Reader	An entertaining computer-generated, hyperreali...	2008-03-10 00:00:00	9559	Toy Story

Clean the data. Retain only cases where fresh and quote are present and non-empty. Remove repeated observations (there are such in data). How many cases will be left?

```
In [98]: rotten = rotten[rotten.fresh != "none"]
         rotten = rotten.drop_duplicates()
         print("Number of cases:", len(rotten))
```

```
Number of cases: 12823
```

Select training and validation data. You would like to split all cases into 80-20 groups. However, as before, this may be slow. Start slow with perhaps 100 random quotes split into two groups.

Your task is to nd the closest training quotes for each test quote, and based on those predict if the movie is fresh or rotten according to the quote.

Follow the same steps you did with the texts above:

(a) create dictionary and BOW of all quotes

(b) run k-NN with several dierent k-s, and predict if fresh or rotten. In each time compute the accuracy.

(c) transform your data into TF-IDF form and repeat k-NN.

(d) inspect a few cases where the tomato was correctly/incorrectly predicted. Can you explain why the algorithm behaved in the way it behaved?

```
In [101]: ntrain = 1500
          nVal = 300

          train_df = rotten.sample(n=ntrain, random_state = 123)
          val_df = rotten.loc[~rotten.index.isin(train_df)].sample(n=nVal, random_state = 123)

          X_train = train_df.quote
          X_val = val_df.quote
          y_train = train_df.fresh
          y_val = val_df.fresh

          sentences = np.concatenate((X_train.values, X_val.values),axis=0)

          vectorizer = CountVectorizer(min_df=0)
          vectorizer.fit(sentences)
          X = vectorizer.transform(sentences).toarray()
```

Now that we've created dictionary and BOW of all quotes, run k-NN with several dierent k-s, and predict if fresh or rotten. In each time compute the accuracy.

** Essentially, we are doing the same thing as what we did above!**

```
In [103]: # BOW
          train_cases = X[:ntrain]

          for l in [1, 5, 25]:
              start = time.time()
              knn(l)
              end = time.time()
              print("Time elapsed:", end - start)
```

```
k: 1
Accuracy: 1.0
Time elapsed: 21.175053119659424
k: 5
```

```
Accuracy: 0.7533333333333333
Time elapsed: 21.226906061172485
k: 25
Accuracy: 0.6866666666666666
Time elapsed: 19.433835983276367
```

Transform your data into TF-IDF form and repeat k-NN.

```
In [104]: # TD-IDF
          tf = X.T #term frequency

          idf = np.log(tf.shape[1] / ( 1 + sum(tf != 0) ))
          idf = np.diag(idf)

          tfidf = np.dot(tf,idf).T

          print(tf.shape)
          print(idf.shape)
          print(tfidf.shape, '\n')

          # normalize
          tfidf = tfidf/np.sqrt(np.sum(tfidf**22))

          # train datasets
          train_tfidf = tfidf[:ntrain]

          for l in [1,5,25]:
              start = time.time()
              knn_tfidf(l)
              end = time.time()
              print("Time elapsed:", end - start)
```

```
k: 1
Accuracy: 1.0
Time elapsed: 107.86344408988953
k: 5
Accuracy: 0.75
Time elapsed: 104.58056592941284
k: 25
Accuracy: 0.6866666666666666
Time elapsed: 108.30084800720215
```

Finally, comment your results. What worked better: reviews or text pages? What worked better: BOW or TF-IDF

We can say once again that: both of our models are best when k = 1, with an accuracy of 1.0. Accuracy decreases as we increase k. BOW and TF-IDF seemed to have similar accuracies with the same k's again, just as we saw in the last section. When we compare reviews vs text pages, we

see that with reviews, accuracy decreases along with k at a lesser rate than text pages, showing us that reviews may have worked better.