CS5600/6600: F20: Intelligent Systems
Assignment 11
Numerical NLP: Searching USENET News Groups

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 20, 2020

1 Learning Objectives

1. Vector Space Model of Information Retrieval
2. Vocabulary Normalization
3. Texts as Bags of Words
4. Text Similarity Metrics

Introduction

As usual, this assignement consists of two problems – a paper analysis (Problem 1) and a coding problem (Problem 2). The main objective of the coding segment of this assignment (Problem 2) is to give you some exposure to numerical NLP methods which we discussed in the past two lectures (on 11/09 and 11/11). Another objective is to give you additional tools to work with digital text datasets in your final projects for this class or other projects in the future. Quite a few of you chose to work with various text datasets for your final projects, which was a pleasant surprise to me. Incidentally, I use the term *digital text* in a very broad, inclusive sense, which doesn't confine it to NL texts. DNA datasets are also digital text datasets and the techniques of covered in Problem 2 are as applicable to them as they're to NL datasets.

Problem 1 (3 points)

My original intent was to assign you a technical paper on numerical NLP methods that discusses how stemming, stoplisting, and clustering are used on large text datasets. This changed two days ago when Jarom Allen, a student in this class, sent me a link to "The Computational Limits of Deep Learning" by Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, Gabriel F. Manso (the pdf's included in the zip). This paper made my semester. Thank you so much, Jarom! This is what university education is all about – open knowledge sharing and open, respectful discussion. I learn as much, and often more, from my students, as, I hope, they learn from me.

I swallowed this paper in half an hour and then re-read more carefully to enjoyed it even more. I wish I had sufficient time to re-read it a dozen times. It resonated very deeply with me because of my own thoughts and reflections over the past decade. In several of my publications, I made cautious warnings, not so much *against* DL as *about* it, based on my observations from my electronic beehive monitoring research: the cost of data storage and training DL models for embedded systems becomes prohibititely expensive to justify better accuracy. Yes, I can train a better DL model by running it on my GPU for a month on several TB of data stored on a few external storage devices to obtain a $\approx$5 percent improvement over a random forest that trains in a day. More importantly, sometimes I cannot even deploy a trained DL model on an embedded computer (e.g., a raspberry pi) due to the latter's RAM limitations. In a couple of recent research talks, I made a conceptual argument – DL approaches don't even attempt to address (insomuch as they play the radical version of Turing's Immitation Game) the fundamental questions posed by cognitive science and symbolic AI: 1) how is our memory organized?; 2) how we retrieve knowledge relevant to what we're doing and apply it?; 3) how do we represent and share knowledge we have with others through natural language (NL); 4) how and when do we modify our knowledge?; 5) how and why do we forget?. I can go on, but I hope you get the gist of what I'm trying to say.

This paper takes a different approach. It presents mathematical/statistical arguments and observations (and even elements of a mathematical theory) on the ever growing computational requirements of DL and pursues some implications. If one, rather cautiously, extrapolates forward the reliance of bulldozing on ever growing GPU farms, one logically concludes that progress along these lines may well be economically, technically, and environmentally unsustainable.

We cannot take these findings lightly or brush them away as pessimistic nonsence, because this investigation is a comprehensive study where the authors have analyzed 1,058 research papers on various DL methods and applications in five different domains (image classification, object detection, question answering, named entity recognition, and machine translation) touted by DL proponents as the best testimony to the validity, correctness, and promise of bulldozing.

For your paper analysis, read only the first 18 pages. You don't have to read, unless you want to or have sufficient time, to read the supplementary materials. The paper is a great review of the data-driven intelligence part of this course and contains a superb and well-documented list of references. You don't have to agree with my views. I accept and welcome alternative views (so long as they are well articulated and argued). I always look forward to your analyses and enjoy reading and commenting on them.

And, don't worry that you didn't get to read a paper on vocabulary normalization and clustering. Problem 2 emphasizes all the important technical aspects of vocabulary normalization. Text clustering will have to wait until after the Thanksgiving break.

Problem 2 (2 points)

Let's build a mini search engine to retrieve posts from 20 USENET newsgroups from `sklearn.datasets`. An advantage of this dataset is that it's readily available in sklearn. Install it from the link in the previous sentence if you don't have it on your computer. Another advantage is that these are real, unfiltered posts. A disadvantage is that the language in some posts is foul and offensive. On the other had, if we want to be objective researchers, we wouldn't want it any other way – there's no substitute for raw, real-world data. You may want to review quickly the pdfs of the second part of the 11/09 lecture and the entire 11/11 lecture to brush up on the terminology.

Before we dive into the source code, a quick side note inspired by a few email exchanges with a student on the previous coding assignment. I write in LaTeX and pdf my documents with `pdflatex`. If you copy and paste straight from my pdfs, watch out for single and double quotation marks. If Python or Lisp complain about those characters, you'll have to re-type them.

Loading USENET Data

A text-based search engine starts with an *indexed corpus*, a vector space where each document is converted to a *feature vector* or a *feat vec*. Sometimes the indexed corpus is referred to as a *feature map* or a *feat mat*, for short. I'll use these terms henceforth. I'll also refer to features as *feats*.

Our first taks is to index the USENET data. In other words, we need to convert texts into bags of words and then convert those bags of words into feat vecs. Here's how we load the USENET data. The second call may take below may take a few seconds, depending on the speed of your internet connection, because it downloads 14MB of data.

```
>>> import sklearn.datasets
>>> usenet_data = sklearn.datasets.fetch_20newsgroups()
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 MB)
```

Here's how we can get the raw text of a newsgroup post.

```
>>> usenet_data.data[500]
'From: bjorndahl@augustana.ab.ca\nSubject: Re: document of .RTF\nOrganization:
Augustana University College, Camrose, Alberta\nLines: 10\n\nIn article
<1993Mar30.113436.7339@worak.kaist.ac.kr>, tjyu@eve.kaist.ac.kr (Yu TaiJung)
writes:\n> Does anybody have document of .RTF file or know where I can get it?\n> \n>
Thanks in advance. :)\n\nI got one from Microsoft tech support.\n\n--
\nSterling G. Bjorndahl, bjorndahl@Augustana.AB.CA or bjorndahl@camrose.uucp
\nAugustana University College, Camrose, Alberta, Canada (403) 679-1100\n'
```

Let's get the names of the USENET groups we'll work with.

```
>>> usenet_data.target_names
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware',
'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles',
'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med',
'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
'talk.politics.misc', 'talk.religion.misc']
```

The function `load_usenet_data()` in `find_usenet_groups.py` (it's in the zip) loads the data. To run the source code in `find_usenet_groups.py`, you need to install `sklearn` and NLTK.

```
def load_usenet_data():
    """ Load USENET data """
    print('Loading USENET data...')
    usenet_data = sklearn.datasets.fetch_20newsgroups()
    assert len(usenet_data.target_names) == 20
    print('USENET data loaded...')
    return usenet_data
```

Let's load the data and put it in the variable.

```
>>> from find_usenet_posts import *
>>> usenet_data = load_usenet_data()
Loading USENET data...
USENET data loaded...
```

Normalizing USENET Data with Stemming and Stoplisting

We have the raw data now. On to vocabulary normalization! We need to define a class to normalize the vocabulary with stemming and stoplisting. The class `StemmedCountVectorizer` below is one way to achieve this goal. This vectorizer uses the snowball stemmer from `NLTK` and the stoplist available in `CountVectorizer`, one of the `sklearn` vectorizer classes. You can also use the Porter stemmer (the link and code samples are in the 11/11 lecture pdf).

```
from sklearn.feature_extraction.text import CountVectorizer
import nltk.stem
english_stemmer = nltk.stem.SnowballStemmer('english')
class StemmedCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        analyzer = super(StemmedCountVectorizer, self).build_analyzer()
        return lambda doc: (english_stemmer.stem(w) for w in analyzer(doc))
```

The function `vocab_normalize_usenet_data()` in `find_usenet_groups.py` uses `StemmedCountVectorizer` to normalize the vocabulary and return the feat mat that maps each post into its feat vec where feats are the stems of the words that were not stoplisted.

```
def vocab_normalize_usenet_data(usenet_data):
    vectorizer = StemmedCountVectorizer(min_df=1, stop_words='english')
    feat_mat = vectorizer.fit_transform(usenet_data.data)
    ## the next two lines are for debugging purposes only.
    num_samples, num_features = feat_mat.shape
    print('number of posts: {}, number of features: {}'.format(num_samples, num_features))
    return vectorizer, feat_mat
```

Let's run it. This may take take a short while.

```
>>> vectorizer, usenet_data_feat_mat = vocab_normalize_usenet_data(usenet_data)
number of posts: 11314, number of features: 110992
>>> type(usenet_data_feat_mat)
<class 'scipy.sparse.csr.csr_matrix'>
```

Since it takes a while to compute the feat mat of 11,314 posts, it makes sense to pickle for re-use. This can be done with `pickle_usenet_feat_mat()` in `find_usenet_groups.py` as follows.

```
>>> pickle_usenet_feat_mat(usenet_data_feat_mat, 'usenet_data_feat_mat.pck')
```

To load the pickled feat mat into Python, use `unpickle_usenet_feat_mat()`.

```
>>> fm = unpickle_usenet_feat_mat('usenet_data_feat_mat.pck')
>>> type(fm)
<class 'scipy.sparse.csr.csr_matrix'>
>>> fm.shape
(11314, 110992)
```

To sum up, in order to index the USENET dataset, we need to

1. load the data;
2. define a vectorizer;
3. do the vocabulary normalization;
4. pickle the feat mat;
5. pickle the vectorizer.

Let's repeat these 5 steps in Python.

```
>>> from find_usenet_posts import *
>>> usenet_data = load_usenet_data()
Loading USENET data...
USENET data loaded...
>>> vectorizer, feat_mat = vocab_normalize_usenet_data(usenet_data)
number of posts: 11314, number of features: 110992
>>> pickle_usenet_feat_mat(feat_mat, 'usenet_feat_mat.pck')
>>> pickle_usenet_vectorizer(vectorizer, 'usenet_vectorizer.pck')
>>> exit()
```

Let me note that the above steps generalize to *any* text dataset. Of course, our vocabulary normalization procedures will depend on a specific dataset. For example, we can use the above vocabulary normalization tools (without any modification) on a twitter dataset. However, a DNA dataset will have its own vectorizer and stoplister. Specifically, we'll need to decide what DNA sequences we want to use as feats and what sequences we want to stoplist. In other words, we'll need to implement custom vectorizer and stoplister classes which will be based on our domain knowledge.

Another important thing to note is that *feat maps and vectorizers go hand in hand.* You may have asked yourself why we've saved the vectorizer in step 5. Saving the feat mat makes perfect sense, but why the vectorizer? Because we cannot make sense of the feat mat unless we know and have access to the vectorizer that computed it. We can construct multiple vectorizers to *fit* to the same text dataset. Each time we fit a vectorizer to a dataset, we construct a feat mat. When a user query comes in, we need to use a vectorizer that was used to construct a specific feat mat to extract the feats from the user query and match it against that feat vecs in the feat mat (i.e., to find the feat vecs close to the user query's feat vec).

Let's load the pickeled feat mat and the vectorizer back Python and investigate what the feat mat's structure.

```
>>> from find_usenet_posts import *
>>> vectorizer = unpickle_usenet_vectorizer('usenet_vectorizer.pck')
>>> feat_mat = unpickle_usenet_feat_mat('usenet_feat_mat.pck')
```

Recall that we have 11 314 feat vecs each of which consists of 110 992 feats. Let's confirm this.

```
>>> feat_mat.shape
(11314, 110992)
```

How do we access individual feat vecs? Straightforward! Each feat vec is a row in the feat mat. Let's access feat vec 8668.

```
>>> feat_vec_8668 = feat_mat.getrow(8668)
>>> feat_vec_8668
<1x110992 sparse matrix of type '<class 'numpy.int64'>'
with 65 stored elements in Compressed Sparse Row format>
```

This format isn't too helpful, is it? No worries! We can convert it into an array.

```
>>> feat_vec_8668 = feat_mat.getrow(8668).toarray()
>>> feat_vec_8668
array([[0, 0, 0, ..., 0, 0, 0]])
>>> feat_vec_8668.shape
(1, 110992)
```

Hey, this is way better, isn't it? The next question is – how do we acctually access which of the feats are present in the feat vec and which are absent. All we need to realize is that `feat_vec_8668` is a binary vector where 0 means that the corresponding feat is abscent and 1 – that it's present. We'll use the `numpy.where()` function. Look it up in the numpy documentation if you've never worked with it before.

```
>>> import numpy as np
>>> abscent_feats, present_feats = np.where(feat_vec_8668 == 1)
>>> present_feats
array([  5649,   6124,   6512,  10360,  16674,  23284,  23286,  25506,
        26215,  28435,  32430,  40718,  40906,  41114,  41842,  42928,
        43079,  43806,  46866,  54543,  63276,  65841,  65988,  70004,
        70415,  72697,  75033,  75036,  75055,  78143,  78380,  79985,
        82002,  82613,  82713,  82726,  90369,  90526,  90777,  94652,
        95211,  96298,  98825, 101486, 104737, 106259])
```

OK. This array gives us the *numbers* of feats that the vectorizer found in this newsgroup post. We're very close, but not quite there yet. The next thing is to figure out the names of each of these feats (i.e., the actual terms computed by the stemmer from the words that were not stoplisted).

This is where the vectorizer comes in handy again, because we fit it to this text corpus to construct the feat mat. Now we can now use to map each feat number in `present_feats` to the feat name (i.e., a word stem). Here's how.

```
>>> vectorizer.get_feature_names()[82726]
'probabl'
>>> vectorizer.get_feature_names()[94652]
'subject'
>>> vectorizer.get_feature_names()[106259]
'write'
```

In other words, feat 82726 is 'probabl', feat 94652 is 'subject', feat 106259 is 'write', etc. We can speed it up with list comprehension.

```
>>> feat_names = vectorizer.get_feature_names()
>>> present_feat_names = [feat_names[i] for i in present_feats]
>>> present_feat_names
['174850', '19', '1993apr2', '29', '6289', '___', '____', 'acd4', 'agre',
'articl', 'bottl', 'darken', 'day', 'dealership', 'deterg', 'do', 'don',
'dump', 'experi', 'hell', 'kmart', 'line', 'littl', 'mean', 'merri',
'mount', 'necess', 'necessarili', 'need', 'open', 'organ', 'paul', 'poorer',
'price', 'prm', 'probabl', 'sedv1', 'send', 'sez', 'subject', 'suspici',
'tank', 'true', 'use', 'way', 'write']
```

So these are the actual feats that the vectorizer mapped to numbers. What's the actual text of the post that the vectorizer mapped to this feat vec? Easy! All we need to do is load the raw usenet data and and retrieve post 8668. We already know how to do it.

```
>>> from find_usenet_posts import *
>>> usenet_data = load_usenet_data()
Loading USENET data...
USENET data loaded...
>>> usenet_data.data[8668]
  'From: jwg@sedv1.acd.com ( Jim Grey)\nSubject: Re: Necessity of fuel injector cleaning
  by dealership\nOrganization: Hell                                          \nLines: 19\n\n
  In article <1993Apr2.174850.6289@cbnewsl.cb.att.com> prm@cbnewsl.cb.att.com (paul.r.mount)
  writes:\n>\n>In your experience, how true is it that a fuel injector cleaning\n>will do much
  more good than just using detergent gas.   While I\n>agree that a clogged fuel injector would
  darken my day, how clogged\n>do they get, and is $59 a good price (or can I do it myself by
  buying\n>a can of ____ (what?) and doing ___ what?\n\n\nA "fuel injector cleaning" at the
  dealer is probably little more than\nthem opening your gas tank, dumping in a bottle of
  fuel injector cleaner,\nand sending you on your merry way $59 poorer.  Go to KMart and buy
  the\ncleaner yourself for $1.29.\n \nJust because you dealer sez you need it, don\'t mean
  it\'s necessarily so.\nBe suspicious.\n \njim grey\njwg@acd4.acd.com\n'
```

So, in the opinion of a certain Jim Grey from the organization called Hell, a fuel injector cleaning works much better than using detergent gas. Furthermore, says Jim Grey, 'just because you dealer sez you need it, don't mean it's necessarily so. Be suspicious.'

If you look that our vectorizer that uses the snowball stemmer and the standard stoplist extracted such features as `'174850'`, `'19'`, `'1993apr2'`, `'29'`, `'6289'`, `'___'`, and `'____'`. I don't think that feats like this are particularly useful. Such feats often lead to unexpected matches in that completely unrelated documents are found to be closest to user queries. But, we won't work on getting rid of them in this assignment and stick with the vectorizer we've defined above. I just wanted to point out that there's room for improvement in our vectorizer.

Text Similarity Metrics

The next question to the vectorization of user queries and the definition of text similarity metrics that match the user query's feat vec with each feat vec in the feat mat. Let's work on these. I've defined two simple vector similarity functions `find_usenet_groups.py` – `euclid_dist(v1, v2)` and `norm_euclid_dist(v1, v2)`. The first function computes the euclid distance between the two vectors; the second one computes a normalized version of the euclid distance. You can define your own similarity functions (e.g., cosine, dot product, logical binary and, manhattan distance, etc.).

Let's define a user query and construct a feat vec from it with our vectorizer.

```
>>> usr_q = 'is fuel injector cleaning necessary?'
>>> usr_q_fv = vectorizer.transform([usr_q])
>>> usr_q_fv
<1x110992 sparse matrix of type '<class 'numpy.int64'>'
with 4 stored elements in Compressed Sparse Row format>
>>> usr_q_fv.toarray()
array([[0, 0, 0, ..., 0, 0, 0]])
>>> usr_q_fv.shape
(1, 110992)
```

We know the list comprehension trick to get the actual feat names from the constructed user query's feat vec. Let's use it.

```
>>> feat_names = vectorizer.get_feature_names()
>>> usrq_present_feat_names = [feat_names[i] for i in usrq_present_feats]
>>> usrq_present_feat_names
['clean', 'fuel', 'injector', 'necessari']
```

So the vectorizer extracted four feats from the user query: `'clean'`, `'fuel'`, `'injector'`, and `'necessari'`, which makes sense.

We're finally in the position to use our similarity metrics to compute the similarity b/w the user query and a specific feat vec in the USENET dataset's feat map.

```
>>> usr_q = 'is fuel injector cleaning necessary?'
>>> euclid_dist(vectorizer.transform([usr_q]), feat_mat.getrow(8668))
12.609520212918492
>>> norm_euclid_dist(vectorizer.transform([usr_q]), feat_mat.getrow(8668))
1.0167190417606449
```

Finding Posts Relevant to User Queries

Implement the function `find_top_n_posts(vectorizer, user_query, doc_feat_mat, dist_fun, top_n=10)`. The stub is in `find_usenet_posts.py`. This function takes a vectorizer object, a user query string, a feat mat computed with the vectorizer, a distance function, and the `top_n` keyword parameter that specifies how many posts to retrieve.

This function first computes the feat vec of the user query. Then, for each feat vec v in the feat map, it uses the distance function to compute the similarity coefficient between the user query's feat vec and v. Each similarty coefficient is saved in a dictionary that maps the number of the feat vec in the feat map (i.e., the row number in the feat mat) to its similarity score with the user query feat vec. The dictionary is converted to a list of key-val 2-tuples. The list is sorted by the second element of each 2-tuple from smallest to largest and the `top_n` first elements of the sorted list are returned. I should note that the sorting order depends on our text similarity metric. Since in the case of the euclid distance the smaller the score, the better, we're sorting from smallest to largest. Other similarity metrics may sort in the opposite order.

Here's a sample run of my implementation. It may take a few seconds, depending on how fast your hardward is.

```
>>> find_top_n_posts(vectorizer, usr_q, feat_mat, norm_euclid_dist, top_n=5)
user query: is fuel injector cleaning necessary?
user query feat vector:
   (0, 37315) 1
  (0, 49840) 1
  (0, 58285) 1
  (0, 75035) 1
Searching USENET posts..
Searching over...
[(8668, 1.0167190417606449), (11185, 1.0281273426612174), (5337, 1.077032961426901),
(3763, 1.3038864705364368), (4355, 1.3192362830276845)]
```

Remember that, in order to see the actual texts, you need to load the raw data and then access the corresponding post by its number.

```
>>> usenet_data = load_usenet_data()
Loading USENET data...
USENET data loaded...
>>> usenet_data.data[8668]
  'From: jwg@sedv1.acd.com ( Jim Grey)\nSubject: Re: Necessity of fuel injector cleaning
  by dealership\nOrganization: Hell                                         \nLines: 19\n\n
  In article <1993Apr2.174850.6289@cbnewsl.cb.att.com> prm@cbnewsl.cb.att.com (paul.r.mount)
  writes:\n>\n>In your experience, how true is it that a fuel injector cleaning\n>will do much
  more good than just using detergent gas.   While I\n>agree that a clogged fuel injector would
  darken my day, how clogged\n>do they get, and is $59 a good price (or can I do it myself by
  buying\n>a can of ____ (what?) and doing ___ what?\n\n\nA "fuel injector cleaning" at the
  dealer is probably little more than\nthem opening your gas tank, dumping in a bottle of
  fuel injector cleaner,\nand sending you on your merry way $59 poorer.  Go to KMart and buy
  the\ncleaner yourself for $1.29.\n \nJust because you dealer sez you need it, don\'t mean
  it\'s necessarily so.\nBe suspicious.\n \njim grey\njwg@acd4.acd.com\n'
```

We can optimize the retrieval by distributing it over a cluser of computers, each of which contains the feat map for a subset of the USENET data. In the ideal case, one computer per group. But, we can also make it more efficient by clustering the data with K Means, matching user queries against the centroid vectors, and then searching within each cluster. We'll do this part in the next assignment *after* the Thanksgiving break.

What to Submit?

1. `cs5600_6600_f20_hw11_paper.pdf` with your one page analysis of the paper;
2. `find_usenet_posts.py` with your impelementation of `find_top_n_posts()`.

Happy Reading, Writing, and Hacking!