Day 4: Text Processing

Overview

Today we will discuss text processing. Our exercises will be grounded in an email dataset (either yours or Kenneth Lay's). After today, you will be able to compute the most important terms in a particular email, find emails similar to the one you are reading, and find people that tend to send you similar emails. And since emails are just text documents, this applies to any text document you can think of, like webpages, articles, or tweets.

As opposed to the data we've seen so far, we will need to clean the data before we can extract meaningful results.

The techniques we will learn today include

- TF-IDF
- Regular Expressions and data cleaning
- Cosine Similarity
- N-gram Analysis

This is the longest lab so far, with 10 exercises. Feel free to skip some of the optional exercises.

Setting Up

Dataset: Kenneth Lay's Emails

Unzip Kenneth Lay's (Pre-bankruptcy Enron CEO) emails that were made public after the accounting fraud scandal.

```
dataiap/datasets/emails/kenneth.zip
```

The full dataset (400MB zipped, >2.4G unzipped) can be found here. Don't download it now: we won't need it until day 5.

Reading the Emails

You will need some code to parse and read the emails. `dataiap/datasets/emails/lay-k/` contains a bunch of folders like `_sent` (sent folder), `inbox`, and other folders depending how Kenneth organized his emails. Each folder contains a list of files. Each file corresponds to a single email.

We have written a module that makes it easier to manage the emails. To use it add the following import

```
import sys
```

```
sys.path.append('data/resources/util/')
import email_util
```

The module contains a class `EmailWalker` and a dictionary that represents an email. `EmailWalker` iterates through all the files under a directory and creates a `dict` for each email file in the directory.

`Email` dictionary object keys

- `folder`: name of the folder the email is in (e.g., inbox, _sent)
- `sender`: the email address of the sender
- `sendername`: the name of the sender (if we found it), or ''
- `recipients`: a list containing all emails in the `to`, `cc`, and `bcc` fields
- `names`: a list of all full names or '' found in senders, to, cc, and bcc list.
- `to`: list of emails in `to` field
- `cc`: list of emails in `cc` field
- `bcc`: list of emails in `bcc` field
- `subject`: subject line
- `date`: `datetime` object of when the email was sent
- `text`: the text content in the email

For example, if you receive an `email_one` dictionary, you can retrieve the sender email by typing `email_one [sender]`.

`EmailWalker` class

The `EmailWalker` class is initialized with the root directory (`dataiap/datasets/emails/lay-k`) that contains email folders (e.g., inbox, _sent). `EmailWalker` is an iterator, and can be conveniently used in a loop:

```
walker = EmailWalker('dataiap/datasets/emails/lay-k')
for email_dict in walker:
    print email_dict['subject']
```

Folder Summaries

Let's extract the key terms of each folder. This should give us a good list of terms that summarize each folder. Computing these terms could also be used to direct a keyword search to the most appropriate folder.

Although the walkthrough will compute key terms for folders, you could also compute it for each contact in your inbox, by each month, or on an email-by-email basis. Additionally, we will only be using the words in the email body, and ignore the subject lines. We'll see if that was a good idea later, and if not, why not!

Term Frequency (TF)

One intuition is that if a term is relevant to a folder, then the emails in the folder should use that term very often. We can count the number of times each term occurs in each email, and the top occurrences of terms across all emails in each folder should best represent the folder.

```
import os, sys, math
sys.path.append('dataiap/resources/util') # fix this path to work for you!!!!
from email_util import *
from collections import Counter, defaultdict

folder_tf = defaultdict(Counter)

for e in EmailWalker(sys.argv[1]):
    terms_in_email = e['text'].split() # split the email text using whitespaces
    folder_tf[e['folder']].update(terms_in_email)
```

The above code iterates over all the emails and splits the message bodies. It then retrieves the `Counter` for the email's folder (`folder_tf[e['folder']]`), and increments the counter for each term in the email. By the end of this loop, we should have term frequency values for each folder. Something similar to (ignore the actual values):

```
'inbox'     --> { 'conference': 10, 'to': 40, 'call': 20, …}
'sec_panel' --> { 'meeting': 10, 'sec': 20, … }
```

Now we can iterate through each of the items in `folder_tf`, sort the counter, and print the top 20 terms for each folder.

```
for folder, counter in folder_tf.items():
    print folder
    sorted_by_count_top20 = sorted(counter.items(), key=lambda (k,v): v, reverse=True)[:20]
    for pair in sorted_by_count_top20:
        print '\t', pair
```

But if we take a look at the output (the top 20 most frequent terms in each folder), they are non-descriptive terms that are simply used often. There are also random characters like `>`, which are clearly not words, but happen to pop up often. You should see something like the following (it's ok if it's not exactly the same).

```
sent
```

```
    ('the', 2529)
    ('to', 2041)
    ('and', 1357)
    ('of', 1203)
    ('in', 905)
    ('a', 883)
    ('>', 834)
```

This suggests that we need a better approach than term frequency, and that we need to clean the email text a bit. Let's do both of these things.

Term Frequency - Inverse Document Frequency (TF-IDF)

Instead of the most popular terms, what we want are popular terms in *this* email or folder that are not popular in *all other* emails or folders. For example, "the" would be considered background noise because it is found multiple times in nearly every single email, so it is not very descriptive. Similarly, "enron" is probably not very descriptive because we would expect most emails to mention the term.

TF-IDF is a widely used metric that captures this idea by combining two intuitions. The first intuition is Term Frequency, and the second is Inverse Document Frequency. These concepts are usually described in terms of abstract "documents."

1. TF: we want to increase a term's weight if it occurs often in a document
2. IDF: we want to decrease a term's weight if it's found in most documents

Notice that the first is calculated on a per-document basis, and the second is across all documents.

The IDF of a given term is formally computed as

```
IDF(term) = log( total # documents / # documents that contain term )
```

This function decreases for a term that frequently occurs in many documents, and high-IDF terms will be the ones that are used in a small set of documents. In our case, a "document" is all the emails in a folder. Thus the numerator is the total number of folders and the denominator is the number of folders where some email in the folder contains the term. Finally, the TF-IDF is simply a multiple of the two values:

```
TF * IDF
```

Now let's write some code to construct a dictionary that maps a term to its IDF value. This code should extend the term frequency code you wrote in the previous section. Fill in the last part to calculate the tf-idf.

The first thing we want to do is compute the number of folders that contain each term. To do this let's first compute the list of terms in each folder:

```
terms_per_folder = defaultdict(set)
nemails = 0
for e in EmailWalker(sys.argv[1]):
    terms_in_email = e['text'].split() # split the email text using whitespaces

    # this collects all of the terms in each folder
    terms_per_folder[e['folder']].update(terms_in_email)
```

The above code reads each email dictionary, extracts the words using `e['text'].split()`, and adds it to the per-folder set (`terms_per_folder[e['folder']]`). We used a `set` to remove duplicate terms. Now our job is to count the number of folders that contain each term.

Each iteration retrieves the terms for a given folder, and adds them all to the counter.

```
allterms = Counter()
for folder, terms in terms_per_folder.iteritems():
    # this will increment the counter value for each term in `terms`
    allterms.update(terms)
```

Great, now we have a dictionary, `allterms`, that maps each term to the number of folders it's in. Now let's actually compute the idf. Notice that we add `1.0` to the denominator to avoid divide by zero errors and so that the denominator is a float. Python truncates integers by rounding down, so if the numerator and denominator are both `int`s, you could end up with a lot of zeros (e.g., 1/2 = 0). The log of 0 is undefined.

```
idfs = {}
nfolders = len(terms_per_folder)  # the number of keys should be the number of folders
for term, count in allterms.iteritems():
    idfs[term] = math.log( nfolders / (1.0 + count) )
```

Finally, you computed the term frequencies in the previous section (`folder_tf`), so let's combine that with our `idfs` dictionary to compute the tf-idf!

```
tfidfs = {} # key is folder name, value is a list of (term, tfidf score) pairs
for folder, tfs in folder_tf.iteritems():
    #
    # write code to calculate tf-idfs yourself!
    #
    pass
```

If we combine `idfs` with each folder's `tf` value, we would compute the `tf-idf`. If we print the top values for each folder, we would see something like (you may not have exactly the same results and that's ok! We used curated your dataset to be a little less "boring" by removing uninteresting folders.):

```
sec_panel/
    ('<<SEC', 0.06234689439353398)
    ('trips,', 0.046513459474442624)
    ('<<Discussion', 0.046513459474442624)
    ('23.doc>>', 0.046513459474442624)
    ('McKinsey&Co.', 0.046513459474442624)
    ('(212)583-5000.', 0.046513459474442624)
    ('May.doc>>', 0.046513459474442624)
    ('Members.doc>>', 0.046513459474442624)
    ('RealNetworks,', 0.046513459474442624)
    ('rsvp,', 0.046513459474442624)
```

As we can see, it's so-so, but the terms don't seem very meaningful, and a lot of them contain characters like `>`! I think we can do better, so let's deal with the noisy data next.

Regular Expressions and Data Cleaning

The email dataset is a simple dump, and each file contains the email headers, attachments, and the actual message. In order to see sensible terms, we need to clean the data a bit. This process varies depending on what your application is. In our case, we decided that we want

1. We don't care about casing. We want "enron" and "Enron" to be the same term.
2. We don't care about really short words. We want words with 4 or more characters.
3. We don't care about stop words. We pre-decided that words like "the" and "and" should be ignored.
4. Reasonable words. These should only contain a-z characters, hyphens, and apostrophes. It should also start and end with an a-z character. That way we don't get `"To:,"` `"From:,"` or `"<<Discussion."`

Let's tackle each of these requirements one by one!

1-2. Casing and Short Words

We can deal with these by lower-casing all of the terms and filtering out the short terms.

```
terms = e['text']lower().split()
terms = filter(lambda term: len(term) > 3, terms)
```

3. Stop Words

The `email_util` module defines a variable `STOPWORDS` that contains a list of common english stop words in lower case. We can filter out terms that are found in in this list.

```
from email_util import STOPWORDS
terms = filter(lambda term: term not in STOPWORDS, terms)
```

4. Reasonable Words (Regular Expressions)

Our final only-real-words requirement is more difficult to enforce. One way is to iterate through the characters in every term, and make sure they are valid:

```
arr = e['text'].split()

terms = []
for term in arr:
    valid = True
    for idx, c in  enumerate(term.lower()):
        if (idx == 0 or idx == len(term)-1):
            if (c < 'a' or c > 'z'):
                valid = False
                break
        elif (c != "'" and c != "-" and (c < 'a' or c > 'z')):
            valid = False
            break
    if valid:
        terms.append(term)
```

This is a pain in the butt to write, and is hard to understand and change. All we are doing is making sure each term adheres to a pattern. Regular Expressions (regex) is a convenient language for finding and extracting patterns in text. We don't have time for a complete tutorial, but we will talk about the basics.

Regex lets you specify:

- Classes of characters. You may only care about upper case characters, or only digits and hyphens.
- Repetition. You can specify how many times a character or pattern should be repeated.
- Location of the pattern. You can specify that the pattern should be at the beginning of the term, or the end.

It's easiest to show examples, so here's code that defines a pattern of strings that start with either `e` or `E`, followed the characters `nron`. `re.search` checks if the pattern is found in `term` and returns an object if the pattern was found, or `None` if the pattern was not found.

```
import re
term = "enronbankrupt"
pattern = "[eE]nron"
if re.search(pattern, term):
    print "found!"
```

The most basic pattern is a list of characters. `pattern = "enron"` looks for the exact string "enron" (lower case). But what if we want to match `"Enron"` and `"enron"`? Writing `re.search('enron', term) or re.search('Enron', term)` would suck. That's where character classes come in!

Brackets `[ ]` are used to define a character class. That means any character in the class would be matched. You can simply list the individual characters that are in the class. For example `[eE]` matches both "e" and "E". Thus `[eE]nron` would match both "Enron" and "enron". `[0123456789\-]` means that all digits and hyphens should be matched. We need to escape (put a backslash before) "-" within `[ ]` because it is a special character.

It's tedious to list individual characters, so `-` can be used to specify a range of characters. `[a-z]` is all characters between lower case "a" and "z". `[A-Z]` are all upper case characters. `[a-zA-Z]` are all upper or lower case characters. There are other shortcuts for common classes. For example, `\w` (without the brackets) is shorthand for `[a-zA-Z0-9]`. Note that we didn't escape the `-` because it specifies a range within `[ ]`.

`[a-z]` only matches a single character. We can add a special characters at the end of the class to specify how many times it should be repeated:

- `?`: 0 or 1 times. For optional characters
- `*`: 0 or more times.
- `+`: 1 or more times
- `{n}`: exactly `n` times
- `{n,m}`: between `n` and `m` times (inclusive).

For example, `[0-9]{3}-[0-9]{3}-[0-9]{4}` matches phone numbers that contain area codes. Again we don't't escape the `-`'s because it specifies a range within `[ ]` and is not interpreted as a range outside the `[ ]`. This pattern fails if the user inputs "(510)-232-2323" because it doesn't recognize the `( )`. Can you modify the pattern to optionally allow `( )`?

Finally, `^` and `$` are special characters for the beginning and the end of the text, respectively. For example `^enron` means that "enron" must be at the beginning of the string. `enron$` means that the "enron" should be at the end. `^enron$` means the term should be exactly "enron".

Great! You should know enough to create a pattern to find "reasonable words", and use it to re-compute the `tfidfs` dictionary and print the 10 most highly scored terms in each folder!

5. Make a data cleaning function

It helps to create a function that performs all of the data cleaning for us. Thus we created a function called `get_terms( )`:

```
def get_terms(message_text):
    terms = message_text.split()
    #
    # all the cleaning from this section
    #
```

and replaced the code:

```
terms_in_email = e['text'].split()
```

with the code:

```
terms_in_email = get_terms(e['text'])
```

If you get stuck, take a peek in `dataiap/day4/get_terms.py`. The file has an example of regular expressions and an implementation of `get_terms()`.

Exercise 1: Compute IDF differently (optional)

IDF is simply one method to normalize the Term Frequency value. In our case we computed IDF on a per-folder basis. However we could just as easily compute the IDF value on a per-email basis. In this case, we would count the total number of emails, and the number of emails that contain a term.

Compute the IDF using this method and see what the pros and cons are. We found that computing it on a per-folder basis can dramatically reduce the IDF score even if only one email in each folder contained the term. The per-email basis avoids this, but causes the top TF-IDF values of a lot of the folders to overlap.

Exercise 2: Compute per-sender TF-IDF (optional)

Remember how tf-idf is defined for abstract documents? So far, we've defined a "document" as an email folder. Now change your code to compute TF-IDF on a per-sender basis. The email dictionary contains a key `sender` that contains the email address of the email's sender.

We recommend copying the file you have so far so you don't lose it!

Cosine Similarity

If you did the last exercise, you have a per-person description of everyone who sent email at Enron. With Cosine Similarity, we can find other people that have sent similar emails.

Let's now figure out which folders are most similar to each other. That would be nice to see how Kenneth is grouping his emails. Cosine similarity is a common tool to achieve this.

The main idea is that folders that share terms with high tf-idf values are probably similar. Also, they if they share lots of terms then they are probably similar.

Let's say we have a total of 1000 terms across all of the email senders. Every folder has a tf-idf score for each of the 1000 terms (some may be 0). We could model all of the scores of a folder as a 1000-dimensional vector, where each dimension corresponds to a term, and the distance along the dimension is the term's tf-idf value. The cosine of the two email senders' vectors measures the similarity between them. Suppose the vectors were A and B. Then the cosine would be:

```
cos(A,B) = (A·B) / ((||A|| * ||B||) + 1.0)
```

The numerator is the sum of all the tf-idf terms the senders have in common. The denominator is the product of the vector norms. Once again, we add `1` in case either vector is 0.
A `cos(A,B)` of 1 means they are identical and 0 means the senders are independent from each other (the vectors are orthogonal).

Here is how we would calculate the cosine similarity of two folders, using the `tfidfs` dictionary you computed in the previous section. We assume that `tfidfs` is a dictionary where each value is a list of `(term, tfidf-score)` pairs

```
from math import *
sec_scores = dict(tfidfs['sec_panel'])
fam_scores = dict(tfidfs['family'])

# loop through terms in sec_scores
# if term also exists in fam_scores, multiply both tfidf values and
# add to numerator
numerator = 0.0
```

```
for sec_key, sec_score in sec_scores.iteritems():
    dotscore = sec_score * fam_scores.get(sec_key, 0.0)
    numerator += dotscore

# compute the l2 norm of each vector
denominator = 0.0
sec_norm = sum( [score**2 for score in sec_scores.values()] )
sec_norm = math.sqrt(sec_norm)
fam_norm = sum( [score**2 for score in fam_scores.values()] )
fam_norm = math.sqrt(fam_norm)
denominator = sec_norm * fam_norm + 1.0

similarity = numerator / denominator
```

This computes the similarity between the `sec_panel` folder and the `family` folder. Can you modify the above code to compute similarities between every pair of folders? Which ones are most similar? It will help to first modify the above code into a function

```
def calc_similarity(folder1_tfidfs, folder2_tfidfs):
    # compute the similarity between the two arguments
    #
    # your code here
    #
    return similarity
```

Then we could call the function with `sec_panel` and `family` and get the same answer:

```
calc_similarity(dict(tfidfs['sec_panel']), dict(tfidfs['family']))
```

Exercise 3: Most similar folders

Extend the above code to rank the folders by how similar they are, where similarity is the cosine score. When computing similarity, we typically took the top 100 tf-idf terms in each vector instead of all of the terms.

Exercise 4: Most similar senders (optional)

Now modify the code you have written so far to compute the cosine similarity between every pair of email senders. You can do this later, because it can take a long time. Alternatively, you can pick a few of the senders and just calculate it for them.

N-grams

Notice that we have used the word "term" instead of "word" when describing tf-idf and cosine similarity. This is on purpose.

The problem with using a single word, as we have, is that we lose the context of a word. For example, "expensive" could be part of the phrase "not expensive," or part of the phrase "very expensive." These have opposite meanings, but we bunch both together when using a single word for the term.

One popular way to add more context is to simply use more than one word per term. We could instead use 2-grams, which are all 2 consecutive word sequences. For example, the phrase "she ate the turtle" has the following 2-grams:

```
she ate
ate the
the turtle
```

An n-gram is an n consecutive word sequence.

Exercise 5: 1 and 2-grams (optional)

Re-run one of the analyses using 1 and 2 grams. How well did that work?

Exercise 6: Email Subject

So far we have extracted the terms from the email body text (`e['text']`). We found that the results are so-so. When people write email subjects, they tend to be to the point and summarize the email. Try re-running the analysis using the email subject line (`e['subject']`) instead of the email text.

Exercise 7: More Cleaning

When we forward or reply to an email, the email client often includes the original email as well. This can artificially boost the TF-IDF score, particularly if the email chain becomes very long. The email usually looks like this:

```
ok.  10:40 to be safe:P

On Fri, Jan 6, 2012 at 5:15 PM, Eugene Wu wrote:

> i have a feeling that breakfast foods stop at 11 at clover because thats
> how it works at the truck.
> so maybe 1045 or something is better.
>
```

```
>
> On Fri, Jan 6, 2012 at 5:09 PM, Adam Marcus wrote:
>
>> see you there at 11!
>>
>>
>> On Fri, Jan 6, 2012 at 5:05 PM, Eugene Wu wrote:
>>
>>> lets have brunch at 11.  That way we skip the rush as well.
>>>
```

Do some more data cleaning to remove the email copies before computing TF-IDF. Hint: lines starting with `>`.

Exercise 8: Normalizing Weights

In our current version of TF-IDF, a person's terms will be artificially boosted if he/she sends you a ton of emails. This is for two reasons

1. He or she will likely have a term vector with a larger variety of terms.
2. He or she will have larger term counts that everyone else.

This will cause the person to have high cosine similarity with a majority of the people in your mailbox. Fix this problem by dividing a person's TF value by the l2norm of the term vector.

The l2norm of a vector can be computed as:

```
import math
vect = [1, 2, 3, 4] # list of numbers
vect = [ item * item for item in vect ]
total = sum(vect)
math.sqrt(total)
```

Exercise 9: Removing Names (optional)

You'll find that names of people end up with very high tf-idf scores often due to signatures. Although it's correct, we want to find people that send similar email content (e.g., topics) so we would like non-name terms. The email dictionary objects contain fields called `sendername` and `names` that store english names. Add everyone's first name and last name to our list of stop words.

We found that this improved our results when we analyzed our gmail emails.

Exercise 10: Analyze Your Emails (optional)

We have written a script (`dataiap/resources/download_emails.py`) that you can use to download your own email over IMAP. However before you can run it, make sure you installed the following python modules:

- dateutil
  - PIP users can type `sudo pip install python-dateutil`
- pyparsing
  - PIP users can type `sudo pip install pyparsing`

Now run the script to see the parameters

```
python download_emails.py -h
```

You can pass optional imap address, username, and password parameters, otherwise it will default to gmail's imap address and prompt you for the missing username and password. It will create the folder `./[YOUR EMAIL]/` and download your email folders into that directory. If you have a lot of emails, it can take a long time. We strongly recommend doing this after class. If everyone runs this in class, it may overload the wireless.

See if you can uncover something interesting!

Done!

Today, you learned the basics of text analysis using an email dataset!

- We used TF-IDF to give each term in our emails a weight representing how well the term describes the email/folder/sender.
- We found that text documents require significant amounts of data cleaning before our analyses make sense. To do so we
  - Computed and removed stop words
  - Used regular expressions to restrict our terms to "reasonable" words
  - Removed copied email content (from replied-to/forwarded emails)
  - Normalized tf-idf values
- We used cosine similarity to find similar folders and senders

We've only touched the surface of text-analysis. Each of the components we've discussed (tfidf, cleaning, and similarity) are broad enough for courses to be taught on them.