Homework Assignments

Information Retrieval 1 [2015/2016]

Module 2: Ranking Models (2pts)

Deadline: Monday, January 18th, midnight
Submit: An IPython Notebook with the necessary (a) **implementation**, (b) **explanations**, (c) **comments**, and (d) **analysis**. Code quality, informative comments, detailed explanations of what each block in the notebook does and convincing analysis of the results will be considered when grading.
Filename: <your id>-hw2.ipynb

The homework will cover the following topics covered in Lecture 1, 3 and 4:

- Language Models;
- Distributional Semantics.

In this homework you will build a small scale search engine from scratch. The methods you will use here are the basic techniques used by all commercial search engines before the uprise of Google. Google only amended these techniques to incorporate the web link structure, and the user clicks.

For the purpose of this homework you will work with a small document collection comprising 39,580 news articles from the Financial Times, and the Los Angeles Times. For the purpose of testing the quality of your search algorithms you will use a set of 50 queries, and a set of relevance judgements provided in a qrel file. Each line in the qrel file has the following structure:

```
query dummy docno relevance
```

Hence for each query and document you can find the relevance label of that document in the qrel file.

Step 1: Implement NDCG [5%]
In order to be able to evaluate your search algorithm implement the normalized discounted cumulative gain at cut-off rank 10, nDCG@10.

Step 2: Implement the randomization significance test [5%]
To compare the quality of two search algorithms, you can compute the mean NDCG@10 over all the queries in the query set and calculate the difference between the NDCG@10 for algorithm A and the NDCG@10 for algorithm B. However, as discussed in Lecture 1, difference can be due to randomness of the queries, and hence a statistical significance test should be run over the NDCG@10 values of the two algorithms. For the purpose of this homework implement the randomization test. Use 100 iterations for the outer loop. The randomization test will only be used in Step 10.

Step 3: Preprocess the document corpus and queries [0%]

The document corpus comes in a specific structure. To get an idea on the structure inspect the file provided. Each document is included between <doc> and </doc> tags, while the interesting tags inside each document are the <headline> and the <text> tags. A simple way to parse the document collection is by using BeautifulSoup in python, but you can use any method you want. Some example code to load the documents can be found below - but you can also write your own, better code:

```
from bs4 import BeautifulSoup

def getDocuments(filename):
        infile = open(filename)
        raw = infile.read()
        infile.close()
        soup = BeautifulSoup(raw, "lxml")
        for doc in soup.findAll("doc"):
            headline = maintext = ""
            if doc.find('headline'): headline = doc.find('headline').text
            if doc.find('text'): maintext = doc.find('text').text
```

You are ready now to preprocess and clean the data. For instance, you may want to remove punctuation, numbers and stopwords. Further some of the documents contain html tags. You may also want to further use BeautifulSoup to remove those. To remove punctuation and numbers, you can simply use regular expressions, re, although there may be better ways to deal with that. You can further remove stop-words such as “a”, “the”. Conveniently, there are Python packages that come with stop word lists built in. You can import a stop word list from the Python Natural Language Toolkit (NLTK). You'll need to install the library if you don't already have it on your computer; you'll also need to install the data packages that come with it. This will allow you to view the list of English-language stop words. Further, you can reduce words to their stems using e.g. Porter stemmer or Lemmatizing (both available in NLTK) which will allow you to treat "messages", "message", and "messaging" as the same word. Some very basic example code (which can be easily extended for better preprocessing) can be found below:

```
class Preprocessor(object):
    def __init__(self, stemmer):
        self.stemmer = stemmer

    def __stem_tokens(self, tokens):
        stemmed = []
        for item in tokens:
            stemmed.append(self.stemmer.stem(item))
        return stemmed

    def preprocess(self, text):
        lowers = self.text.lower()
        letters_only = re.sub("[^a-zA-Z]",
                      " ",
                      lowers)
        tokens = nltk.word_tokenize(letters_only)
        filtered = [w for w in tokens if not w in stopwords.words("english")]
        stemmed = self.stem_tokens(filtered)
        return stemmed
```

Using the code above it took me 3891 seconds (~1hr) to load and preprocess the document collection.

Step 4: Construct an Inverted Index [0%]

Having reduced each document to a list of stems, you can create an inverted index that allows an easy retrieval of all the documents that contain a word **w**. An inverted index can be implemented naively as a Dictionary (in python). The keys of the dictionary are the stems in the collection, and the values are lists of pairs [docno, term frequency] for each document that contain this stem. A sample code to construct an Inverted Index is given below (not tested, and likely inefficient):

```
class IndexNode(object):
    def __init__(self, docno, count):
        self.docno = docno
        self.count = count

class InvertedIndex(object):
    def __init__(self, collection):
        self.index = defaultdict(list)
        for docno in collection.documents:
            document = collection.documents[docno]
            counts = Counter(document.getText())
            for word in counts:
                self.index[word].append(IndexNode(docno, counts[word]))

    def tf(self, words):
        results = defaultdict(list)
        for word in words:
            for node in self.index[word]:
                results[word].append([node.docno,node.count])
        return results

    def df(self, words):
        results = {}
        for word in words:
            results[word] = len(self.index[word])
        return results

    def cf(self, words):
        results = defaultdict(list)
        for word in words:
            collection_freq = 0
            for node in self.index[word]:
                collection_freq = collection_freq + node.count
            results[word] = collection_freq
        return results
```

Using the code above it took me 54 seconds to create the inverted index, and 3 seconds to get the tf, df, and cf of a 3 word query. Note: for steps 3 and 4 it would be easier to store the inverted index and loaded every time you need to run experiments so that you don't need to spend 1 hour loading and preprocessing the original data.

Step 5: Implement Basic Query Likelihood Models [20%]
A large variety of language models has been proposed in the literature. A survey on language models can be found here. The basic language modeling approach (i.e., the query likelihood scoring method) can be instantiated in different ways by varying (1) θ_D, (2) estimation methods of θ_D (e.g., different smoothing methods), or (3) the document prior *p(D)*. Implement a query likelihood scoring method using multinomial language models with the following smoothing techniques:

- Jelinek-Mercer (explore different values of **λ** in the range {0.1, 0.2, …, 0.9})
- Dirichlet Prior (explore different values of **μ** in the range of {500, 1000, … , 3000})
- Absolute discounting (explore different values of **δ** in the range {0.1, 0.2, …, 0.9})

Note: Don't forget to use log computations in your calculations to avoid underflows.

For each smoothing method, run the experiments and compare the different parameter values. Create a plot for each smoothing methods with the x-axis being the parameter value and the y-axis the NDCG@10 of the method. Display your plots as the output of this step.

Choose the optimal values for each parameter and compare the three smoothing methods. Create a table with the 3 comparisons, and report NDCG@10 for each algorithm.

Step 6: Two stage smoothing and estimation of **μ** and **λ** [20%]
Implement a two stage Language Model as described in the lecture and the original paper: http://www.stat.uchicago.edu/~lafferty/pdf/two-stage.pdf (last equation of Section 3). Compute the log of the score to avoid underflows.

Estimate **μ** and **λ** based on the leave-one-out method for **μ** and the Expectation Maximization for **λ**. For **μ**, due to the high complexity of the method choose only a handful of documents, and a handful of terms to leave out, and not the entire collection. Obviously the more documents and the more terms the better but judge pragmatically the computational expense of this step. What matters is to show your work and not the entire scale of things. For **λ** use $p(q_j|C)$ instead of $p(q_j|U)$ for the computations.

Run the experiment for the estimated **μ** and **λ** and compare the results with the methods in Step3. Report your comparisons and conclusions.

Step 7: Implement Positional Language Models [**Extra 0.5pts credit**]
Another variation of language models is to relax the independence assumptions made in the basic model to capture some limited dependency. The positional language model (PLM) proposed in http://sifaka.cs.uiuc.edu/~ylv2/pub/sigir09-plm.pdf defines a language model for each position of a document, and score a document based on the scores of its PLMs. The PLM is estimated based on propagated counts of words within a document through a proximity-based density function, which both captures proximity heuristics and achieves an effect of "soft" passage retrieval. Implement the PLM, all five kernels, but only the Best position strategy to score documents. Use **σ** equal to 50, and Dirichlet smoothing with **μ** the best value discovered in Step 3.

Given the computational complexity of computing, use the Query Likelihood model with Dirichlet smoothing and **μ** the best value discovered in Step 3 to rank documents. Then only consider the top-10 documents and re-score them based on PLM and the five kernels. Based on the new scores re-rank the documents and compute the mean NDCG@10 for each Kernel. Compare the different Kernels and the PLM to the Query Likelihood from Step 3. Report your conclusions.

Step 8: Train word2vec on collection [10%]
You don't need to implement you own word2vec, however for those particularly interested in distributional semantics you can find a good explanation of the parameter learning for word2vec in the following paper, http://arxiv.org/abs/1411.2738, since the original paper is not particularly explanatory. Instead you can use an existing Python implementation of word2vec [gensim]. If you don't already have gensim installed, you'll need to install it. There is a tutorial that accompanies the Python Word2Vec implementation, here. To train the word2vec on the document collection of this homework you will need to re-preprocess the collection so that you provide word2vec with the appropriate input (i.e. sentences).

Although word2vec does not require graphics processing units (GPUs) like many deep learning algorithms, it is compute intensive. Both Google's version and the Python version rely on multi-threading (running multiple processes in parallel on your computer to save time). ln order to train your model in a reasonable amount of time, you will need to install cython (instructions here). word2vec will run without cython installed, but it will take days to run instead of minutes.

Further, there are a number of parameter choices that affect the run time and the quality of the final model that is produced.

- **Architecture**: Architecture options are skip-gram (default) or continuous bag of words.
- **Training algorithm**: Hierarchical softmax (default) or negative sampling.
- **Downsampling of frequent words**: The Google documentation recommends values between .00001 and .001.
- **Word vector dimensionality**: More features result in longer runtimes, and often, but not always, result in better models. Reasonable values can be in the tens to hundreds.
- **Context / window size**: How many words of context should the training algorithm take into account?
- **Worker threads**: Number of parallel processes to run. This is computer-specific, but between 4 and 6 should work on most systems.
- **Minimum word count**: This helps limit the size of the vocabulary to meaningful words. Any word that does not occur at least this many times across all documents is ignored. Reasonable values could be between 10 and 100. Higher values also help limit run time.

Make a choice on the aforementioned parameters, and explicitly mention your choice in your submitted homework.

[**For those interested**] If time and computational power permits experiment on multiple choices, and report your conclusions.

Step 9: Produce a vector representation for documents/queries [20%]
Word2vec produces vector representations for individual words and not for the entire document. Try three different methods to produce document representations:

- Average or sum of the word vectors in the passage
- Cluster words in the document and use the centroid of the most important cluster
    - Use K-means (experiment on different values of K)
    - Use Chinese Restaurant Process can also be used.

[**For those interested**] Alternatively you can train a paragraph2vec model, also implemented in the gensim package. There is a tutorial about paragraph2vec here. Different parameter decisions need to be made here as well.

Step 10: Run experiments with word2vec [5%]
Being able to produce a representation of documents and queries allows you now to run experiments using these representations. Given the possible computational cost of computing document representations, use the Query Likelihood model with Dirichlet smoothing and **μ** the best value discovered in Step 3 to rank documents. Then only consider the top-10 documents and re-score them based on Step 8. Based on the new scores re-rank the documents and compute the mean NDCG@10 for each of the three different methods in Step 8. Compare the performance of the different methods implemented in Step 8 to the Query Likelihood from Step 3. Report your conclusions.

Note that it is important to preprocess queries here the exact same way you preprocessed the collection to train word2vec on.

Step 11: Compare all methods [15%]
Consider the best performing system in Step 3, Step 4, Step 5 (if implemented), and Step 8 and compare them with each other both reporting mean NDCG@10 values and p-values produced from the randomization test. Report your analysis.

Q & A (from emails I've received so far)

Q: In order to implement NDCG I must compute the normalisation factor. The book says it is "calculated to make it so that a perfect ranking's NDCG at k for query j is 1". So since in the qrel file I observe 2 as max relevance, I firstly compute the DCG for 10 documents with max relevance and then I get the norm_factor as its inverse.

A: Well, not exactly. Your way of thinking is correct but by perfect ranking we mean perfect possible ranking. If for example you create DCG@10 using max relevance (e.g. 2) but the query by it's nature only has a single document of relevance 2, 3 documents of relevance 1 and all the rest of the documents of relevance 0, then a perfect possible ranking would be: {2 1 1 1 0 ... 0} since by nature no algorithm can do better than this.

Q: I'm implementing the NDCG and the way I did is like:
1. I sort (descendant) the ranking and calculate the ideal dcg.
2. I calculate the dcg on the original vector
3. Divide dcg/idcg

A: The ideal ranking that you construct only considers the documents that your algorithm has retrieved. If your algorithm ranks the entire collection, then that should be fine. But if your algorithm only returns, e.g. the top-10 or top-100 documents then it is likely that some relevant documents in the collection are not included in the top-10 or top-100. Here is an example:

Ranker 1 returns {2 2 1 0 0 0 0 0 0 0} for query Q1.
Ranker 2 returns {2 2 2 1 0 0 0 0 0 0} for query Q1.

With your computations of NDCG, the NDCG@10 for both of these rankers is 1!!! But that should not be the case.
Let's assume that in the qrels, i.e. is the file with all the relevant documents for Q1 there are 4 documents of relevance 2, and 2 of relevance 1. The ideal ranking in this case should be {2 2 2 2 1 1 0 0 0 0} and it is the same for all Rankers.

Q: The BeautifulSoup LXML parser does not work
A: Install it, http://lxml.de/build.html#building-lxml-on-macos-x