BingBling

Assignment 1 - Relevance Feedback

Team Members

- Nikhil Mitra (nm2868)
- Prakhar Srivastav (ps2894)

Running the program

On CLIC

To run on the *CLIC* machines, just run the following set of commands.

```
$ cd /home/ps2894/bingbling
$ ./run.sh
```

Build it locally

In order to build the project on your computer, just install the python packages the program depends on. A list of these packges is provided in the the `requirements.txt` file. Since this program uses `nltk`, nltk data also needs to be downloaded.

```
$ pip install -r requirements.txt
$ python -m nltk.downloader all
$ ./run.sh
```

Internal Design

The program is divided primarily into four files.

- `config.py`: Contains key-value pairs for configuring the program. This includes the account keys, the output format and configuration parameters for the underlying algorithm.
- `bing.py`: Contains the implementation specific to the bing search engine which is responsible for making the API call, parsing and organizing the results, and then passing those results off to the main program. This has been separated so that another search engine can be added via a drop-in replacement.
- `starter.py`: This file is the main driver program. Its job is to get the results from the search engine module (bing in this case), make changes to the results (such as adding the query in the dataset) and call the public methods exposed by the `corpora` module. `starter.py` takes care of all the user-interaction logic such as taking input from user, calculating the precision, changing the query and lastly, deciding when to stop.

- `corpora.py`: Corpora.py is the primary workhorse of the program. It implements the **Rocchio's Algorithm** for modifying the query. It relies on external libraries such as NLTK to help with stopwords filtering and word tokenization. As a part of Rochhio's algorithm, the code in this file does the following -

  1. Creates a vocabulary
  2. Uses the vocabulary to compute tf-idf for each word
  3. Creates the document vector for each vector
  4. Applies the Rocchio's query update formula
  5. Returns a set of (word, score) tuples sorted in the descending order of scores

Program Flow

The program flow is quite straightforward. The `run.sh` file starts the program which loads up the `starter.py` file. In the `main` function of this file, the user is asked to provide his input query, a precision value and an optional **BING** account key.

After these inputs are provided, a call to the `start` function is made which then calls the search engine (`bing.py`) to fetch the results from the API. On retrieval of these results, the starter programs starts a loop to get the user input in order to separate the relevant and non-relevant documents.

With these indices of relevant and non-relevant documents in hand, the `corpora` module is called which then takes the data, builds up the vocabulary and runs the **Rocchio's Algorithm** on it.

Query Modification

As stated earlier, Rocchio's algorithm is used to provide relevance feedback. Under that model, all docs are treated under the **vector space model**. The formula for updating the query under this paradigm looks as below -

$$\vec{q}_m = \alpha \vec{q}_0 + \beta \frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j - \gamma \frac{1}{|D_{nr}|} \sum_{\vec{d}_j \in D_{nr}} \vec{d}_j$$

In the equation above, the set of D_r and D_{nr} is provided by the user. The program then simply finds each of $\vec{d_j}$ - which is the document vector for each document. To calculate this vector, the program computes the tf-idf score of each of the words that each document is comprised of. If V is the vocabulary, i.e. a set of the all distinct words in the documents then

$$\vec{d_j} = \sum_{w_i \in V} weight(w_i) tfidf(w_i)$$

where $weight(w_i) = 1$ if $w_i \in d_j$, 0 otherwise. It is to be noted, that while building the vocabulary the stop words are culled out since they do not add much value to the results. Additionally, while building the vocabulary, words are lowercased and tokenized using the NLTK's tokenizer. This ensures that a word like `didn't` is tokenized correctly as ("did", "n't") and not ("didn","t").

```python
def _constructVocab(self):
    words = set()
    stopWords = set(stopwords.words('english'))
    for d in self.documents:
        words.update([w for w in d if w.isalpha() and w not in stopWords])
    return words
```

When these document vectors are computed, the formula above can simply be applied for given values of α, β and γ. In our case, the value of these parameters were chosen to be $\alpha = 1$, $\beta = 0.75$, $\gamma = 0.15$ with the understanding that non-relevant documents should be penalized and the relevant ones should be used to bump up the query terms.

The final step then just involves taking the words with top five weights and returning a set of tuples of (word, score).

```
topIndices = vector.argsort()[-5:][::-1]
wordList = sorted(self.bagOfWords)
return [(wordList[i], vector[i]) for i in topIndices]
```

In the `starter.py` module, the logic selecting the top words for appending to the query is written which simply checks the first two words and if the scores of these match. If there's a match, it returns the first two and if there isn't it returns the first word.

```
# filter - choose words that are not in the query already
candidates = [(w, s) for w, s in corpora.getUpdatedQuery() \
              if w not in set(self.query.split())]
(w1, s1), (w2, s2) = candidates[0], candidates[1]
newQueryWords = [w1, w2] if s1 == s2 else [w1]

# build new query
self.query = " ".join([self.query] +  newQueryWords)
```

Bing Account Key

The bing account key is `Byygq1zI2KKyssKp8UvVe3DV/v6Aa0FEsKrE+pqDa0s`

File List

```
|-- README.md
|-- transcripts
|   |-- musk-output.txt
|   |-- gates-output.txt
|   |-- taj-mahal-output.txt
|-- requirements.txt
|-- run.sh
|-- src
|   |-- __init__.py
|   |-- corpora.py
|   |-- config.py
|   |-- bing.py
|    -- starter.py
|-- tests
    |-- __init__.py
    |-- corpora_test.py
```