

A Search Engine for Financial Tweets

Based on a Probabilistic IR model → Latent Dirichlet Allocation for Topic Modeling and Retrieval. Inferred using a Collapsed Gibbs Sampler. Novelty: Sentiment Analysis (meter) on Financial Tweets.

Harivallabha R Mukundhan J Nikhil L
2016B4A70519H 2016B4A70355H 2016B4A70507H

Design Document

- Back-end: Implement a search engine for the financial tweets domain based on a probabilistic information retrieval model.
- Perform topic modeling through Latent Dirichlet Allocation.
- A Collapsed Gibbs sampler for Inference.
- Augment the search results with a sentiment analysis based meter indicating the reliability of a particular tweet.
- Language: Python, R
- Front-end: Similar to the Bing Toolbar. HTML stylized by CSS. Integrated by the battery-included python framework: Django
- Dataset: Kaggle Financial Tweets Corpus. This file contains 16k+ tweets about publicly traded companies (and a few cryptocurrencies) that are tagged with the company they are tweeting about and their symbol as well as more fields.
- [\[https://www.kaggle.com/davidwallach/financial-tweets\]](https://www.kaggle.com/davidwallach/financial-tweets)

We mathematically define the notion of a topic to be a probability distribution over our vocabulary. Having defined this, LDA provides a probabilistic model of how a document of our data is created. This model allows us to use statistical inference algorithms to find estimates for the latent random variables which we are interested in.

In a nutshell, the documents are represented in terms of a distribution over probable topics rather than being identified by, say, the bag of words they contain. This is a form of dimensionality reduction. A small note: The number of topics representing each document is a hyperparameter in our implementation of LDA and is not directly inferred from the document data. After that, we find the topic distribution for the query. Similarity among the topics over the documents and the topics identified for the queries is used to rank the documents and display the results.

$$\beta_1 = \begin{bmatrix} 0.04 \\ 0.03 \\ \vdots \\ 0.0 \end{bmatrix} \begin{matrix} \leftarrow \text{equation} \\ \leftarrow \text{integral} \\ \vdots \\ \leftarrow \text{bread} \end{matrix}$$

1. Choose topics $\beta_1, \dots, \beta_K \sim \text{Dir}(\eta)$, where $\eta \in \mathbb{R}^+$ is a fixed parameter.
2. For $d = 1, \dots, D$: Choose the topic distribution of document d as $\theta_d \sim \text{Dir}(\alpha)$, where $\alpha \in \mathbb{R}^+$ is a fixed parameter that does not depend on d .
 - a. Choose the topic of the n^{th} word, $Z_{d,n} \sim \text{Multinomial}(\theta_d)$.
 - b. Choose the n^{th} word, $W_{d,n} \sim \text{Multinomial}(\beta_{Z_{d,n}})$.

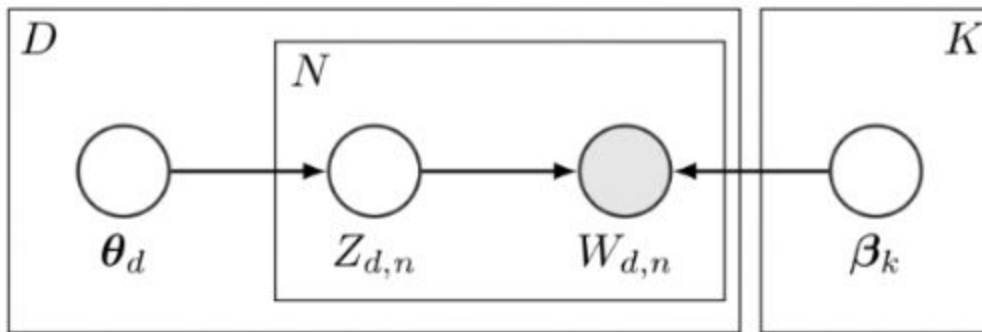



Figure 1: Graphical model of LDA

$$\text{distance}(\hat{\theta}_d, \hat{\theta}_Q) = \sum_{j=1}^K (\sqrt{\theta_{d,j}} - \sqrt{\theta_{Q,j}})^2.$$

Python is a very convenient language, and provides a fast hash-table in the form of a dictionary data structure. The only data structures we need to use in our implementation of LDA for learning topics and displaying relevant results are lists, and dictionaries.

Dictionaries provide $O(1)$ access to values given the key, and lists function like heterogeneous arrays. Python also provides a convenient sort function implemented similar to the `sort()` available in the C++ STL library.

In addition, we've made use of numpy in a couple of places for sampling the multinomial distribution. Numpy has a very efficient random number generator as well.



In addition we've made use of user-defined classes and structs according to the usual good principles of software development in order to have a modular code with well-distributed functionality for readability and easy debugging.

Sentiment Analysis:

We employ the SentimentAnalysis package in R for this feature of the search engine. The preprocessing for the twitter corpus includes: stemming, tokenization and removal of stopwords.

This module works in the following manner:

We tokenize the sentence and categorize the words as positive, negative and neutral using the inbuilt QDAP Dictionary, out of the 4 given choices:

- 1) DictionaryGI: Dictionary with opinionated words from the Harvard-IV dictionary as used in the General Inquirer software
- 2) DictionaryHE: Dictionary with opinionated words from Henry's Financial dictionary
- 3) DictionaryLM: Dictionary with opinionated words from Loughran-McDonald Financial dictionary
- 4) QDAP Dictionary: A subset of the Harvard IV Dictionary containing a vector of words indicating strength

The reason for our choice is that dictionaries 1),2) and 3) give neutral scores for many tweets and the composite dictionary 4) gives us reliable scores on analysis.

The tokenized words are then assigned these score: positive(+1), negative(-1) and neutral(0).

We then add up the scores of all the words and divide by the word count in that tweet, giving us the composite score.

Run-time analysis

1. Preprocessing

```
19.608851194381714  
[21/Oct/2019 12:50:59] "GET /search/?q=thanos+killed+all+that+we+held+dear HTTP/1.1" 200 8402
```

2. Search and Retrieval

```
19.661362409591675  
[21/Oct/2019 12:57:16] "GET /search/?q=thanos+killed+all+that+we+held+dear HTTP/1.1" 200 8402
```

(2) Includes the pre-processing and training time which is the time recorded in (1). Hence, the time taken solely for Search and Retrieval is: (2)-(1) = 0.052512 seconds.

README .txt

- 1) The back-end driver program is called `lda.py` and is stored in: `searchengine\\testscript` directory.
- 2) The dataset is fed into the driver code in line number 209 of `ldatm.py`. This line needs to be changed by the user to suit his/her needs. **Note:** The corpus should be stored as a list of lists. Each list denotes a document, and contains the words in that document.
- 3) The front-end integration occurs in `views.py` and is stored in: `searchengine\\testscript` directory.
- 4) The html and css pages necessary for front end output are integrated with Django and are stored in `searchengine\\templates`. We use a form page based template from django. Thus any user-specific stylizations can be done to `search_results.html`, and `home.html` to change the result display page and the primary search bar page.

Deployment:

- 5) The command to run the project is: `..\searchengine\\manage.py runserver 0.0.0.0:8000`

Note: This command deploys the search engine onto the LAN network which contains the User's PC, and can be accessed by other computers as: `userip:8000`

To deploy just in the user's computer, use: `..\searchengine\\manage.py runserver` in cmd