**Data Extraction(Crawling)**

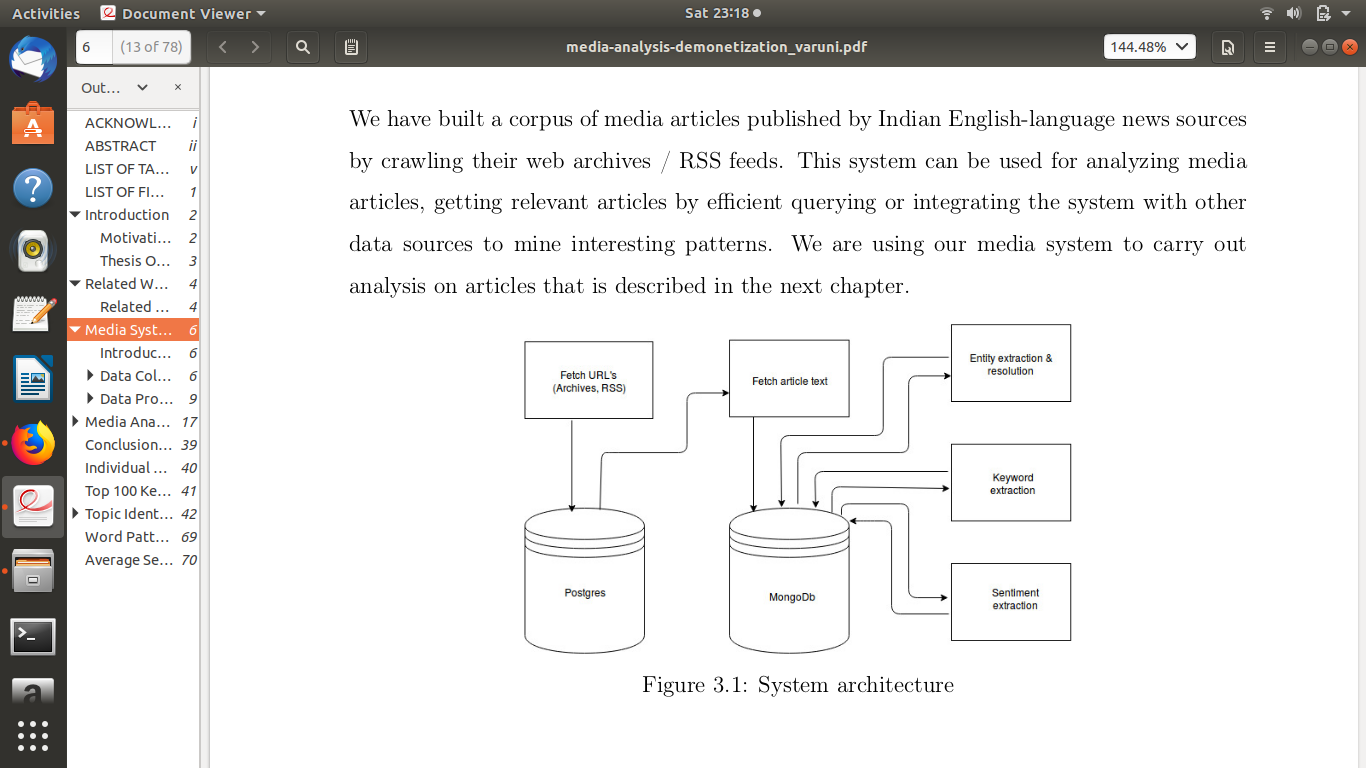
**Introduction to Crawlers**

To find information on the hundreds of millions of Web pages that exist, a search engine employs special software robots, called **spiders**, to build lists of the words found on Web sites. When a spider is building its lists, the process is called **Web crawling**. In order to build and maintain a useful list of words, a search engine's spiders have to look at a lot of pages.

Crawler identifies all of the hyperlinks on the website and adds them to the list of URLs to visit (also known as the crawl frontier). Once a web crawler is given a list of URLs (also called seeds) to check out, it begins visiting each website and downloading the content.

**Crawler functioning**

We have built a corpus of media articles published by Indian English-language news sources by crawling their web archives / RSS feeds. This system can be used for analyzing media articles, getting relevant articles by efficient querying or integrating the system with other data sources to mine interesting patterns.



**Data Collection: Fetching Article URL and Text**

We have collected news articles of categories national, international, regional, sports, opinion and business. This categorization is followed by all news sources in general. National, international and regional are political news articles. Some articles are also tagged as FRONT\_PAGE if this information is available while crawling their urls. Sometimes URLs are published under multiple categories by news source. So we associate an array of categories for each article. Opinions include editorials, regular columns, letters to the editor and special edition columns. Currently, we have data from sources The Hindu, The Times of India, Indian Express, The New Indian Express, Telegraph, DeccanHerald and Hindustan Times.

**Archived URls**

Some news sources had their articles archived online which could be crawled by any user. Archives were available online for all sources present in our system except Hindustan Times. They are crawled into our system archived articles from 2011 onwards . We use Scrapy crawler to crawl archives.

**[Provide the links of these archives, newspaper wise]**

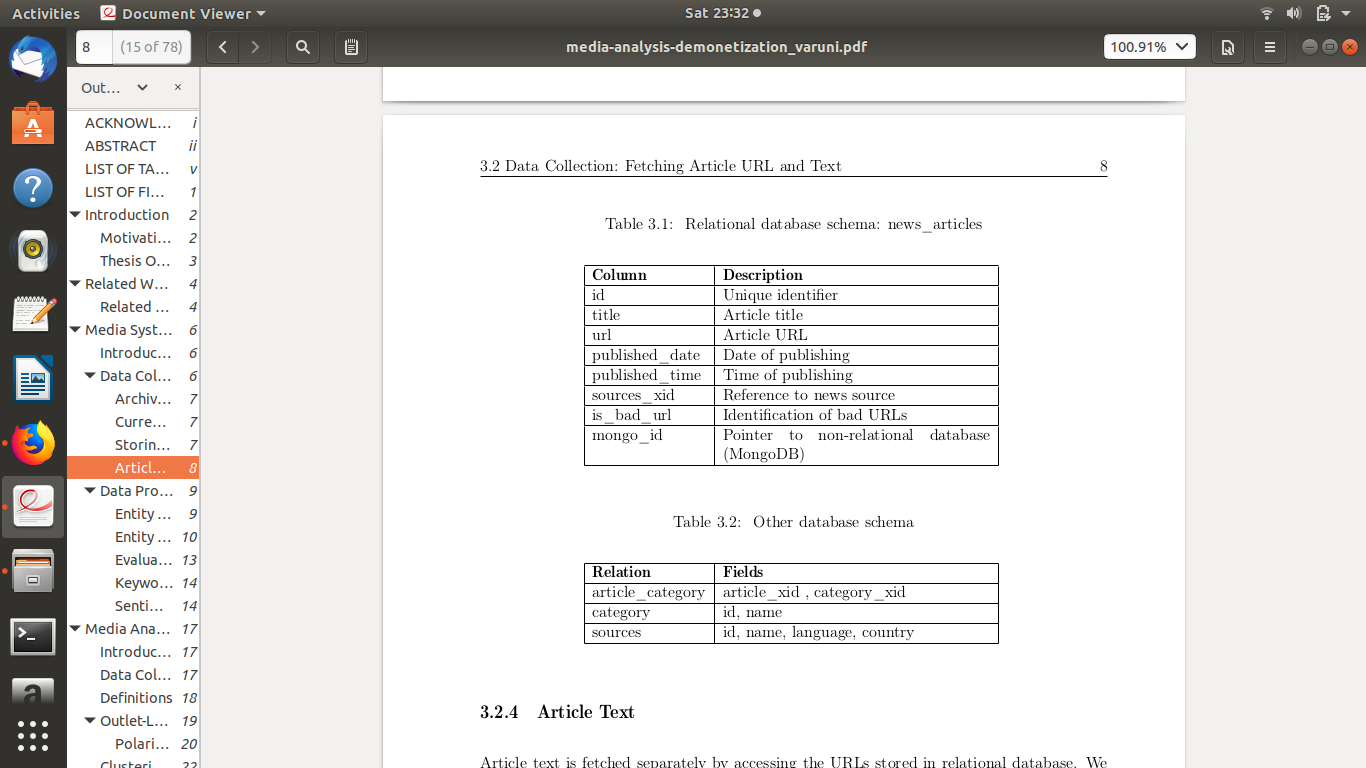
**Current URLs**

News sources publish latest articles through RSS feeds. We have setup an automated script that uses RSSfeedparser to extract current data. It checks for new feeds every five hours **[how? Describe briefly.]**. HTTP header fields like ETag and Last-Modified date are used by some servers as a property of their feeds. ETag is a unique id given to the feeds by the server. If a feed gets updated, value of ETag changes. So when we send these fields as headers, server sends a response only when there is a change in ETag or Last-Modified date of feeds. RSS feeds are available for Hindustan Times ,The Hindu ,Indian Express, New Indian Express, The Times of India only. For the rest of two newspaper i.e. Deccan herald and Telegraph, we use their web archives to fetch new data.

**Storing Article URLs in PostgreSQL**

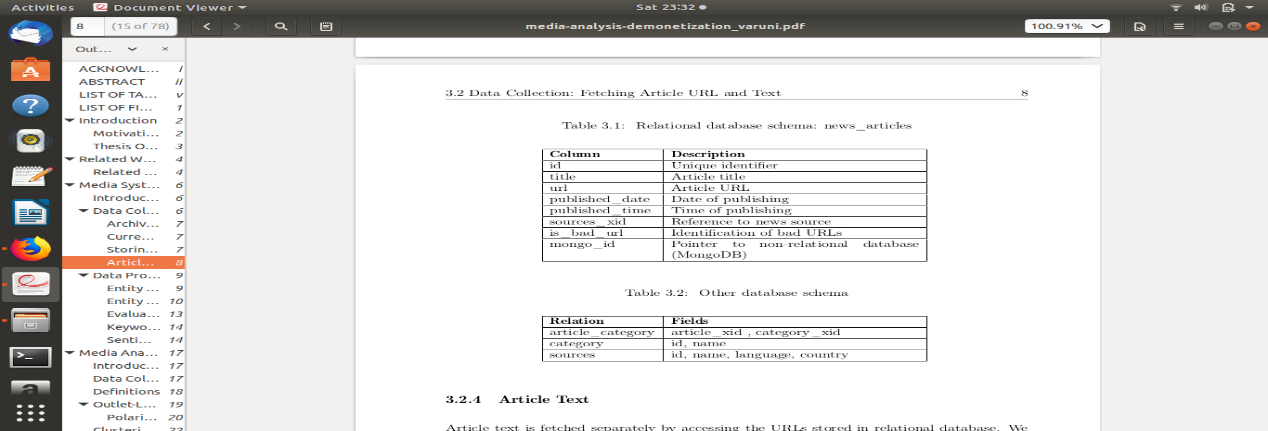
We store meta data of an article URL and information about a news source in a Postgres relational database. News source information like name, language of content, country of circulation is stored. Article information like its title, web link, date of publishing, news source and category is stored. Relational schema of database is given below

***Relational database schema: news\_articles***



**[The figures are mere screenshots. Can this be improved? Almost nothing is visible here.]**

***Other Database Schema***



**Storing of Article Text in Mongodb**

Article text is fetched separately by accessing the URLs stored in relational database. We have used newspaper.py to crawl articles [**location/path?]**. In this process, we fetch text and author information if available. Authors are reporters and columnists. This data is stored in MongoDB which is a no-sql database. An article document in mongodb consists of its meta data which is stored in postgres along with text and author info. An article document in MongoDB has fields:

• articleTitle: Title of article

• articleURL: Web link of article

• categories: Array of categories of article.

• publishedDate: Date of publishing

• publishedTime: Time of publishing

• text: Article text

• author : Author information. Usually reporters and columnists.

• sourceName: News source

• country: Country of circulation of news source

• language: Language of article

• entities: Extracted using Open Calais

• sentiments: Sentiment of Article Text

• keywords: Relevant topics of articles

**How crawlers work for current URLs/RSSFeeds**

**Path to module: /home/gem2/media\_filter/media\_monitor/rssparsers**

**How to run:** Path to media\_monitor$ python rssparsers/startprocess.py

**Python libraries used: feedparser.py**

**Folder structure:**

* **rsssource.py-**Base class inherited by classes that represent RSS URLs of newssources
* **rssfeedurl.py-** Class that abstracts properties of an RSS feed
* **rssfeedparser.py-** Invokes the feedparser to fetch article URLs published by RSS feeds of each source.
* **{news\_source}.py-** A child class that inherits rsssource.RSSSource and abstracts properties of news source.
* **startprocess.py-** Instantiates newssources and invokes the feedparser to fetch URLs

**Working**

Each {newssource}.py file defines a sub class with properties like name of source, language, country of publication, RSS feed URLs for different categories like national, international, regional news, etc. The parse() is overriden in every sub class with different logic to extract published date. Scripts are written for sources HT, Indian Express, TOI, The Hindu and The New Indian Express. startprocess.py instantiates every source sub class and invokes getRSSFeeds() every five hours.

Rssfeedparser.py is the main script which uses feedparser to access the RSS feed URLs and fetch published article URLs. We send HTTP headers like ETag and Last-Modified date while hitting RSS feed URLs so that server sends data only if there are any new articles published from the last time we accessed it. In case of errors because of RSS URLs being changed temporarily/ permanently, appropriate error messages are generated and are handled using exception handling methods.

**How crawlers work for fetching URLs from website**

**Path to module : media\_filter/media\_monitor/scrapy\_crawlers**

**Python libraries used :**

* [Scrapy](https://scrapy.org/) - To crawl archived URLs

**How to run: Path to media\_monitor$ python scrapy\_crawlers/startprocess.py**

**Folder structure:**

* hinduspider.py - A spider to crawl archived URLs of The Hindu
* toispider.py - A spider to crawl archived URLs of TOI (opinions are not available)
* indian\_express.py - A spider to crawl archived URLs of Indian Express
* telegraphspider.py - A spider to crawl archived URLs of Telegraph
* new\_indian\_express.py - A spider to crawl archived URLs of The New Indian Express
* deccan\_herald.py - A spider to crawl archived URLs of Deccan Herald
* middlewares.py - configuration of proxy. Will be used in custom\_settings of all spiders.
* newsitem.py - A class that represents the information to be extracted from web and get stored in postgres
* **startprocess.py - Instantiation of crawler process and assignment of crawling jobs.**

This folder contains the code to crawl URLs from the website of news sources. We are using Scrapy to crawl URLs. In scrapy, a crawler is called spider and it is typically a class. We have to create spiders  (classes) for every news source and invoke the main scrapy process to crawl using individual spiders. So every news source is a class that inherits the base class scrapy.Spider.

A typical spider module in this folder has the following.

* Network and proxy server settings
* Web page URLs (start\_urls) to crawl.
* News article URL to category mappings
* parse() to parse the downloaded web page
* store() to store the news article URLs in postgres.

**Configurations**

All configurations have to be made in startprocess.py. It has an object *spiders,* which contains an object representing each news source.

Each news source object has:

className- Spider class representing source

start- This property has to be filled if we wish to crawl the news source. Eg: If we wish to crawl The Hindu and TOI,New Indian Express, fill in values of *start* and *endTime* (optional)  for the two sources.

endTime- This field is optional. If it is left as None, by default news URLs published from *start* to two days before present day are crawled.

The crawler, after crawling the news sources, sleeps for 24 hours and crawls URLs of next day (two days before the present).

**Article Extraction.**

Articles are crawled continuously and are stored in MongoDB in media-db Database.

**Location of DB** media-db -> Collections ->articles.

After the general article extraction, based on manual set of Keywords, findArticle<eventname>.py script is run. The script extracts the articles based on the set of keywords and stores articles in a different Data base in MongoDB in gem2 server.

**Location of DB** <eventwise>\_media-db-><eventname>\_articles.

**Location of Script** On the <eventname>\_articles dataset script extractkeywordfromarticle<eventname>.py is run to extract top 100 Keywords and to refine the Article extraction. The Script removes irrelevant articles from the dataset and outputs a top 100 Keywords into a txt file using RAKE API. This txt file is manually checked and Keywords fetched are augmented in the set of keywords in the findArticle<eventname>.py and again this script is run. **[the location of the scripts is not mentioned]**

This process continues till the time no new relevant keywords are found.

**Refining Article Extraction.**

The extractkeywordfromarticle<eventname>.py script runs an algorithm to remove irrelevant articles.

The Rules are

1. Frequency of Keywords. If the frequency of keyword is >=2.
2. If keyword is in Top 50% of text. If the keyword is found in the first half of the article.
3. If the keyword is present in select relation (‘amod’,’nmod’,’dobj’,’iobj’,’nsubj’,’nsubjpass’) in Stanford NLP.

If the article satisfies at least one of the above rules then these articles are relevant else are irrelevant and removed.

**[The relationship between the above scripts is not clear. What is the difference between findArticleDemonetization.py and extractkeywordfromarticleDemonetization.py? We need to clarify this further. Also, the events should be extracted using automated keyword augmentation. Which script does that?]**

**LDA**

On the Article collection we run LDA where we try to find out the optimal number of clusters. We map different articles with aspect id.

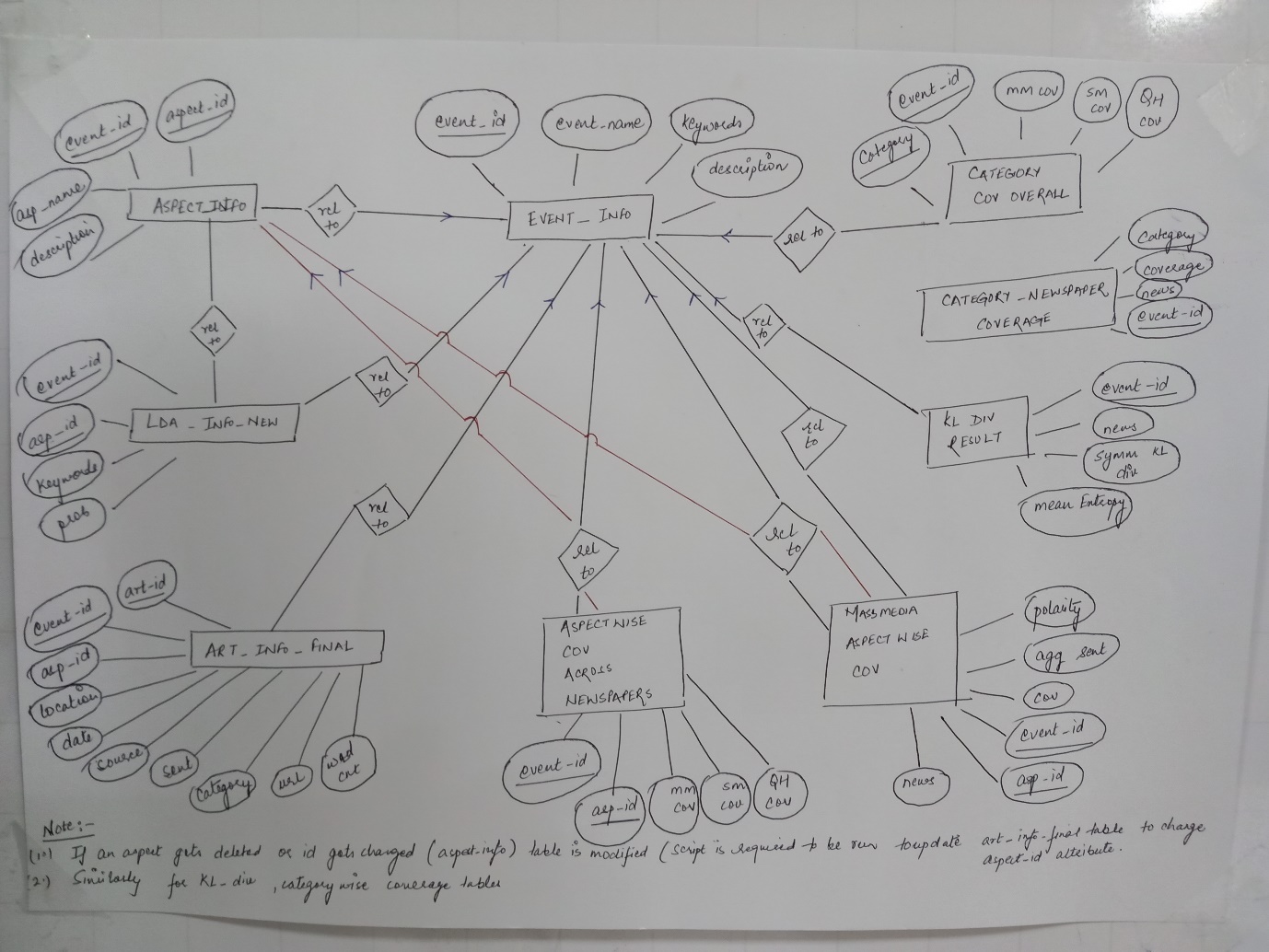
We then map aspect id to aspect name by manually reading the top 20 documents given by LDA. **[Scripts? Locations?]**

**Database.**

The articles from the relevant article dataset <eventname>\_articles is extracted to the gem2 server and stored in postgres SQL server. All the articles are stored in a table.

Location. media\_database (act4dgem.cse.iitd.ac.in) -> Tables -> art\_info\_final.

The below ER diagram explains different tables, schemas and their dependencies.



All the info data is taken from the Postgres table (art\_info\_final) and results are stored in relevant tables in the DB for further visualization.

The description of tables is as below.

1. art\_info\_final. (Master Table) All the articles are stored in this database,Its description is as follows.

|  |  |  |
| --- | --- | --- |
| Column Name | Data Type | Description |
| article\_ID (Primary Key) | Text | Article ID assigned to each article. |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| aspect\_ID | Text | Aspect ID assign to aspect in the event. |
| word\_count | Integer | Length of the article. |
| url | Text | Article URL |
| Category | Text | Category of article |
| vader\_sent | Numeric | Sentiment value in article from tool Vader. |
| vader\_sent\_pos | Numeric | Vader Positive sentiment value in article. |
| vader\_sent\_neg | Numeric | Vader Negative sentiment value in article.  In article(Vader) |
| senti | Numeric | Sentiment value in article through tool Senti strength. |
| senti\_pos | Numeric | Senti Strength positive value in article. |
| senti\_neg | Numeric | Senti strength negative value in article. |
| news\_source | Text | Newspaper name |
| publish\_date | Date | Publish date of the article. |
| Location | Text | Location of article. |

1. event\_info (Master Table). It contains event information.

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| event\_name | Text | It tells the mane of the event. |
| keywords | Text | The augmented set of keywords. |
| description | Text | Description of the event. |

1. Aspect\_info (Master Table). It contains Aspect information.

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| aspect\_ID (Primary Key) | Integer | Aspect ID assign to each aspect in the event. |
| aspect\_name | Text | The name of the aspect. |
| Description | Text | The name of the aspect. |

1. Lda\_info\_new

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| Aspect\_ID (Primary key) | Integer | Aspect ID assign to each aspect. |
| Keywords | Text | The augmented set of top 50 keywords. |
| Probability | Numeric | Probability value for that aspect. |

1. Massmedia\_aspectwise\_coverage. It stores the coverage and sentiment values of media houses.

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| Aspect\_ID(Primary Key) | Text | Aspect ID assign to each aspect. |
| Newspaper | Text | The newspaper name. |
| Coverage | Text | It gives the coverage of the aspect in that event. |
| Agg\_sent |  | It gives the aggregate sentiment value of the aspect in the event. |
| Polarity | Numeric | It gives the polarity value of the aspect |

1. Aspect\_cov\_across\_newspapers. It shows the comparative analysis of different newspaper for mass media, social media and QH data.

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| Aspect\_ID (Primary Key) | Integer | Aspect ID assign to each aspect. |
| Massmedia\_coverage | Numeric | Mass Media coverage of aspect given by a Newspaper |
| Socialmedia\_direct\_coverage | Numeric | Social Media coverage of aspect given by a Newspaper by direct URL Hitting. |
| Socialmedia\_community\_coverage | Numeric | Social Media coverage of aspect given by a Newspaper |
| QH\_coverage | Numeric | Question Hour coverage of aspect given by a Newspaper |

1. Categorywise\_cov\_overall. The details of category wise coverage given by media houses.

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| Category (Primary Key) | Text | Category to which it belong like government, Informal, middle class. |
| Massmedia\_coverage | Numeric | Mass Media coverage of category given by a Newspaper. |
| Socialmedia\_community\_coverage | Numeric | Social Media coverage of category given by a Newspaper. |
| QH\_coverage | Numeric | Question Hour coverage of category given by a Newspaper. |

1. Category\_newspaperwise\_cov

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| Newspaper (Primary key) | Text | The Newspaper name. |
| Category (Primary Key) | Text | The category it belongs. |
| Coverage | Numeric | The coverage given to that category by the newspaper. |

1. KL\_div\_result It shows that for each newspaper which event shows divergence from mean.

|  |  |  |
| --- | --- | --- |
| Column name | Data Type | Description |
| event\_ID (Primary Key) | Integer | Event ID assign to each event. |
| Newspaper (Primary key) | Text | The Newspaper name. |
| Symmetric\_KL\_div | Numeric | The symmetric value of KL divergence. |
| Mean\_entropy | Numeric | The mean entropy value. |

**Entity Resolution**

**Files in use:**

* extract\_entities\_oc.py - Collects the extracted entities from articles, adds some contextual information and dumps into unresolved collection.
* elasticsearch\_oc.py - Resolves entities in unresolved collection creating a new resolved collection.

**[Location of scripts?]**

**Working:**

1. **extract\_entities\_oc.py**

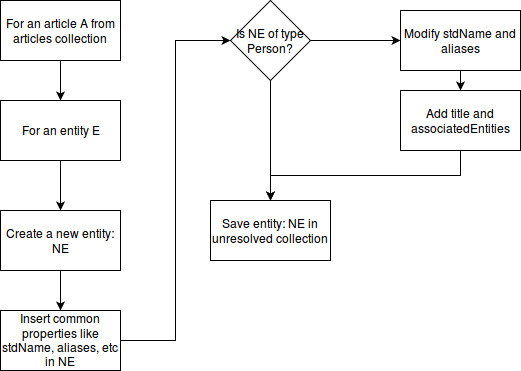
OpenCalais extracts entities given an article. These entities are stored along with article text in an article collection. As a next step, for every entity extracted, we modify their names and add some contextual information extracted from article and store it in unresolved collection. This script does the same.

How to run: python extract\_entities\_oc.py

Configurations:

1. *collection* inside extract(): Name of article collection
2. *collection* inside save(): Name of unresolved entity collection

Overview:



Description:

Only entities of type *'Person', 'City', 'Country', 'Continent', 'Company', 'Organization', 'Region', 'ProvinceOrState'*  are processed further and stored in unresolved entity collection.

An unresolved entity will have the following properties after the execution of this script:

* stdName: Name of entity provided by OpenCalais.
* type: type of entity eg: Person, City, etc  provided by OpenCalais.
* aliases: list of names. Will have stdName as the only element initially. During resolution, this field will hold names of all matching entities.
* articleIds: list of article ids. Initially contains only 1 element which is id of article from which this entity is extracted.
* resolutions: Provided by OpenCalais. Stored for every entity except type *‘Person’*.
* title: List of objects. An object has a phrase and list of articleIds from which the phrase has been extracted. Detailed explanation given below.
* associatedEntities: List of objects. An object has entity name and an associated count. Detailed explanation given below.

For entities of type ‘Person’:

1. Words like Mr, Ms, Mrs, etc are removed from stdName and aliases
2. Words like Chairman, Commissioner, Minister are removed from stdName and aliases and added to *title.*
3. *stdName*s of entities of type Person, Company, Organization extracted from the same article are stored under *associatedEntities.* Each name is associated with a count which is initialized to 1 and incremented every time we come across the same name.
4. *title* is a list of phrases each of which is extracted from the first line where the entity appears in article. It also contains phrases removed in step 2. We identified four patterns to extract title from first sentence containing entity. If a pattern fails, we try to match the next pattern.
   1. Left title: All words in the sentence to the left of entity name with an initial capital letter. Can also include prepositions beginning with small letters.
   2. Right title: All words to the right of entity name with an initial capital letter. Can also include prepositions beginning with small letters.
   3. Weak left pattern: All words (upto 5 words) between ‘said’ and entity name.
   4. Ten words: If all the above patterns fail,  we extract upto 5 words to the left and right of entity name.

**2. elasticsearch\_oc.py**

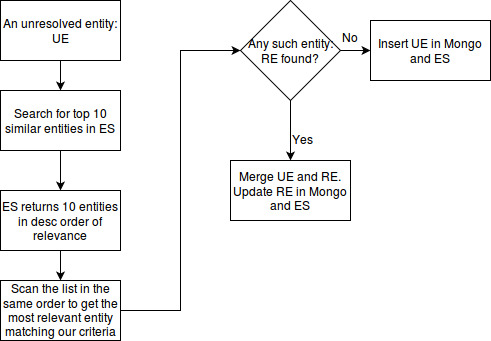
Now we have an unresolved collection. For every entity in unresolved collection, we find the top ten matching entities from resolved collection. From these set of entities, we find a best match. The best matching entity and unresolved entity is merged and resolved collection is updated. Else, if no matching entity is found, unresolved entity is inserted in resolved entity collection. This collection is present in MongoDB and Elasticsearch datastores.

How to run: python elasticsrch\_oc.py **[location?]**

Configurations:

* es\_mapping (declared at the top):  Name of Elasticsearch mapping which is a replica of resolved collection.
* mongo\_coll (declared at the top): Name of resolved mongo collection
* collection (declared at the bottom where the script starts): Name of unresolved mongo collection

Flow diagram of resolution:



**[You also need to mention the pseudocode of our divide and conquer approach of ER. This should explain how the algo works for each chunk – along with how associated entities are processed (the high TFIDF thing). Please refer to Varuni’s original ER algo document (I think this is present in their thesis/system architecture document as well) to see how the pseudocodes are written].**

**[Please also mention the statistics: how much time does it take to resolve x entities, time required to resolve the entire collection, time taken at each step of divide and conquer, memory usage, etc.]**