

Fake News Detection: Fact checking and Bias detection

A Report
*submitted in partial fulfillment of requirements for the
degree of*

Masters of Technology
in Computer Science and Engineering

by

Abhishek Udhav Bagade
Roll No : 163059005

under the guidance of

Prof. S. Sudarshan



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

June, 2019

Dissertation Approval

This is to certify that the thesis titled **FAKE NEWS DETECTION: FACT CHECKING AND BIAS DETECTION** submitted by Mr. **ABHISHEK UDHAV BAGADE (163059005)** to the **Indian Institute of Technology Bombay**, is a bonafide record of the project work done by him and is approved for the award of **Master of Technology in Computer Science and Engineering**.

Examiners



Prof. Soumen Chakrabarti
Dept. of CSE, IIT Bombay



Prof. Kameswari Chebrolu
Dept. of CSE, IIT Bombay

Supervisor



Prof. S. Sudarshan
Dept. of CSE, IIT Bombay

Chairman




Prof. Soumen Chakrabarti
Dept. of CSE, IIT Bombay

Date: 27-06-19
Place: Mumbai

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 27/06/2019


Abhishek Udhav Bagade
163059005

Abstract

The current public discourse is heavily affected by the propagation of Fake News and media. This has severely undermined various societal structures and eroded the trust of general public. The system we plan to build will tackle this disinformation and provide insights to users to make an informed decision about the veracity of the news article.

One approach is to check the given text against a set of articles, users generally use search engines like Google, Bing etc to accomplish this part but these come with a set of restrictions. We create our own data store to overcome these limitations. Once we have information to verify our claims against we can give a verdict about the veracity of the claim.

A given text contains many indicators which may indicate a particular bias towards particular entity. We use Natural language processing and Machine learning to detect these features and check the bias of given text. We use signals from social media to identify clusters of users which show bias towards a particular political entity and use this information as a labelled dataset in supervised classification problem.

We also explore the community characteristics of politically active twitter users to identify source based political bias.

Acknowledgements

I would like to thank my guide **Prof. S. Sudarshan** for always helping me and pointing me in the right direction whenever I was stuck. His guidance is instrumental in the completion of this M.Tech thesis. I would also like to thank **Prof. Soumen Chakrabarti** and **Prof. Kameswari Chebrolu** for their invaluable suggestions and improvements suggested in the work as part of the thesis.

I would also like to thank everyone in the InfoLab, IIT Bombay for their help and guidance and Computer Science and Engineering department for providing the highest quality education and facilities.

Contents

1	Introduction	1
1.1	Previous Approaches	1
1.2	Approach	2
1.3	Fake News Detection system: Previous work	2
1.4	Fake News Detection System Architecture	3
1.5	Hyper-partisanship Detection & Political Bias estimation	4
1.6	Report Plan	5
2	Data: Crawling and pipelines	6
2.1	Scrapy	6
2.1.1	Architecture	7
2.2	FND System Schema	8
2.3	Extracting data	9
2.4	Pipelines	11
2.5	Deployment	12
2.5.1	Available options	12
2.5.2	Approach taken	13
2.5.3	Running archive crawlers	13
2.6	System services	13
2.6.1	Directory structure	13
2.6.2	Maintaining the state of last crawl	14
2.6.3	Virtual Environments	14
2.6.4	Logging	15
2.6.5	Systemd services	15
2.6.6	Cron jobs	16
2.7	Challenges and their solutions	16
2.8	Improvements and tweaks	18
3	Political Bias Detection	20
3.1	Previous Work : Hyper Partisanship detection	21
3.1.1	Semeval 2019 Task 4: Hyper partisan News detection	21
3.2	Political Bias detection using text in Indian Media	22
3.3	Methodology for crawling twitter	23
3.4	Constructing Influence Graph	23
3.4.1	Mention graph	24
3.4.2	Retweet graph	24
3.4.3	Clustering Algorithm	25

3.5 Clustering Results	26
3.6 Organizing and Crawling Data	26
3.6.1 Tweets	26
3.6.2 Cluster IDs	28
3.6.3 News Articles from twitter handles	28
3.7 Text Classification	28
3.7.1 Universal Language Model Fine Tuning	28
3.7.2 Our Setup	29
3.7.3 Results & Discussions	30
3.8 Source Based political bias detection	32
4 Summary and Conclusions	33
Bibliography	34

Chapter 1

Introduction

Fake news has been a large problem in recent times due to the exploitation of social media as a propaganda tool by Governments, Corporations etc. Tackling fake news becomes a major problem due to scale of information exchanged on these platforms. The rise in internet adoption by the general populace increases the scope of such disinformation.

Majority of the cases of propagation of fake news are by actors with malicious intentions but sometimes they are just mistakes on users part. Simply cross checking with credible sources is enough to supply users with enough information to judge the veracity of claims. Generally people confuse names, events and other specific information about an event, these are then propagated to other users through oral or other communication medium like social networking sites. We can easily search the web and find out the true information. This approach works for individual queries but the limits implemented by major search providers like Google make it impossible to use this on a system level. Current fact-checking systems take varied approaches towards solving this problem and we discuss them in next section.

1.1 Previous Approaches

Fact checking systems currently take one of these two approaches

1. In first approach the whole process is automated and is modelled as end to end learning task and the system returns either a score of fake-ness or a binary label(fake or not fake). This type of systems fail to capture all the semantic features of data but do well in some aspects, but one of the most important aspect of identifying fake news is convincing the user why the news article is fake. The explainability of the system is one of the important factors in tackling fake news and needs to be incorporated in any model. Ex. Fake-ometer by IIIT-H
2. In second approach the whole system depends on the manual fact checking of claims, this set of fact checkers are generally experts in related domains like journalists etc. These systems work pretty well as long the choice of experts is unbiased. But this approach ignores the fact that the article itself has many

signals which can be used to identify bias of news articles. Various machine learning models have been proposed which can be used to assist the user make informed decisions. Sub tasks such as Hyperpartisanship detection, target Sentiment analysis etc help the user to Identify if the article is genuine or not. This approach also comes with the problem of multiple sites with different information. Checking across multiple sites becomes a task in itself. Ex. Fact checking sites such as BoomLive etc.

1.2 Approach

We use a combination of both approaches to get better results. We use the already available crowd sourced data on web sites such as <https://www.boomlive.in/>, <https://check4spam.com/> etc. These sites serve as a good repository of fake and genuine articles. We can query against this source of articles and return the verdict. Currently the way people search existing fact checking sources is by using either the search functionality in the web site itself or using a search engine like Google. Google doesn't allow users to query using more than 32 words, this makes matching full length articles impossible. We create a separate collection of articles from multiple fact checking sites to allow the user to use a single service to search over multiple sources.

In case of completely new articles not seen by fact checking sites, we can give related articles matching the given text and all the information extracted by linguistic models. We leave the final decision regarding fake-ness on the user rather than making a judgment. To summarize we do the following to make sure we detect a fake article

1. Collect all the major news articles from relevant sources
2. Find matching news articles for the given text/article.
3. Gain insights from the data available which will help the user verify the credibility of given text.

In this report we mainly tackle first 2 problems. we try to create a comprehensive data set of news articles from which we can query the relevant articles and we design an efficient search and retrieval system. We discuss the architecture of the system and the technical challenges faced in deploying it. The task of extracting insights from the text would be tackled in the next phase.

1.3 Fake News Detection system: Previous work

Previous work done as a part of an internship project crawled news sites using individual scripts written in python. The scripts downloaded the raw HTML page and then extracted data using BeautifulSoup library. These scripts were run regularly using a central script which scheduled as well as executed the crawler runs. The

scripts lacked a uniform structure across all the crawlers and didn't take advantage of common operations performed across crawlers.

We also studied various neural model architectures which performed classification on news datasets. Some of them worked on Stance detection, Hyper partisanship detection etc. but they performed poorly in case of Indian news. We try to construct Indian news specific datasets and train various models on it.

1.4 Fake News Detection System Architecture

The current system has an architecture shown in figure 1.1. We introduce each component briefly next,

1. **Crawlers:** The existing crawlers were supposed to be proof of concept and never meant to scale to the levels we require. We replaced the per site python scripts with Scrapy Framework [1]. This allows to consolidate the whole scraping process into a single framework. This also allows us to use common data processing pipelines for all Scrapers and define a common schema for the extracted data. The advantages of the framework are discussed at length in section 2.1.
2. **Data Pipelines:** The scraped data from crawlers goes through multiple pipelines which perform various operations. The pipelines are described in detail in 2.4. The broad idea of what each Pipeline does is given below.
 - (a) **Validation Pipeline:** Performs basic validation checks on the extracted data and eliminates blank articles.
 - (b) **Stat Aggregation Pipeline:** Collects various statistics about the crawler run and the number of articles crawled and inserts into a Database.
 - (c) **WritetoFile Pipeline:** Dumps the crawled data in JSON files. The files are organized by site name and the topics it belongs to
 - (d) **ImageInsert Pipeline:** Downloads all the images in the articles crawled and indexes it along with metadata in Elastic search[2] back-end.
 - (e) **IndexSolr Pipeline:** Indexes the crawled article in the Apache Solr backend.
 - (f) **NLU Pipeline:** Extracts and stores various information about the article body using IBM Watson NLU [3] API.
 - (g) **OCR Pipeline:** We use Tesseract OCR engine [4] to extract text from images crawled in both English and Hindi language.
 - (h) **Video Pipeline:** We store hashes extracted from videos present in articles in Solr.
3. **Indexing and Search:** We use two different back-end services for indexing and searching our scraped data. We use Apache Solr[5] for text data like article body, URL, etc and Elastic Search[2] for indexing and searching the

Images crawled. We use this information to find if text/article queried is already reported in some news source. Images also constitute a major part in dissemination of fake news. We check if a given image occurs in an article already crawled and the queried context is different than the original context.

4. **Fake News API & Web front-end:** We allow access to our system using a REST-ful API for querying the image and text system separately. The final API combines both into one unified API. As a proof of concept we have a simple web client which returns the result to users against queries.
5. **System services:** We use systemd and cron to schedule the services and ensure they are restarted on failure. We also use separate virtual environments to make sure the dependencies do not clash with each other.

The overall architecture of the system is shown in 1.1. We discuss each of the components in detail in the rest of report

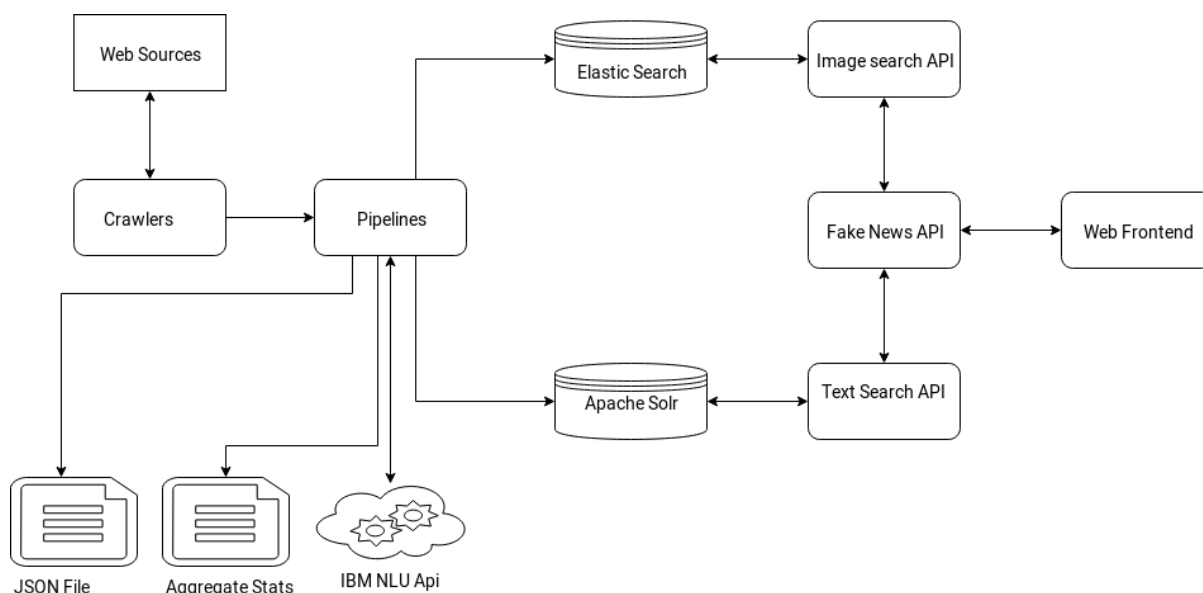


Figure 1.1: Fake news detection system Architecture

1.5 Hyper-partisanship Detection & Political Bias estimation

First phase of our work concentrated on detecting whether the given text is Hyper-Partisan or not. We trained some Machine learning models and got reasonably good results on the Semeval Dataset [6]. The next challenge was using this model in the context of Indian political news. Estimating whether an article is hyperpartisan or not becomes a lot harder in our case as we lack experienced annotators. We try to overcome this problem by reducing it to political bias estimation problem.

Given a text we estimate the political bias of it towards one of three major political entities. We apply various methods to get the labelled data in our context.

We then apply various ML models to see which performs better. We discuss this in detail in section 3

1.6 Report Plan

We discuss the two major aspects of the Fake news detection system we build in this report.

1. In chapter 2 we discuss the system we have and its architecture. We discuss in detail the design decisions we made and the general system structure. This section is supposed to act as a manual for the system.
2. Chapter 3 of report focuses on gathering labelled data and predicting the political bias of given text. We discuss the approaches we take for classifying the given text based on the political bias they show towards a set of political entities.

Chapter 2

Data: Crawling and pipelines

The success of the system depends majorly on the amount of data available to search and match. Majority of the news articles can be easily found in major media sources, cross referencing them with the query gives the user enough information to see if the given text is true or not. This is only possible if we have an extensive store of articles which we can query against.

One more reason in favour of fact checking articles against previously published articles is the rise of fact checking sites such as <https://check4spam.com/>, <https://www.boomlive.in/> etc. Such sites if extensively crawled will solve most of our problems. These sites work by using crowd-sourcing to recognize news articles as fake or genuine. This information is critical to us and we include such sites in our crawl.

Scraping websites comes with its own challenges and there are many approaches one can take. The approach used in the system was supposed to be a proof-of-concept and had many drawbacks as it wasn't supposed to be ready for deployment.

The initial approach used custom python scripts to crawl websites and extract data using python libraries such as Beautiful Soup etc. The problem with such approach is common operations post extraction such as indexing the data, storing it in files etc. becomes difficult to co ordinate among the different crawlers. Many of the tasks are common to crawlers and setting up a data pipeline would solve many of the problems. Also many of the standard crawler operations have to be implemented from scratch which is re-inventing the wheel.

Considering above reasons we choose to migrate to Scrapy Framework[1]. This is a python framework which is developed only for crawling and has many advantages which we'll discuss further in this chapter.

2.1 Scrapy

Scrapy is advertised as an 'An open source and collaborative framework for extracting the data you need from websites. In a fast, simple, yet extensible way.'. The main advantages of scrapy for our project are

1. It reduces a lot of boiler plate code which is required for scraping and provides a very intuitive abstraction to operate with.

2. It provides simple methods to develop a Data pipeline, we can easily write different pipelines to do majority of our tasks
3. Takes care of Data Extraction from the web page by either CSS or XSS rules.
4. Provides default pipeline implementations to download and store multimedia content from web pages like images or videos.
5. Is extensible and open source which makes it easy to change the behaviour of some components to better suit task at hand.
6. Provides settings to bypass scraping protections on news sites.

2.1.1 Architecture

This section gives a brief overview of the architecture of Scrapy, the architecture of scrapy can be understood better through figure 2.1

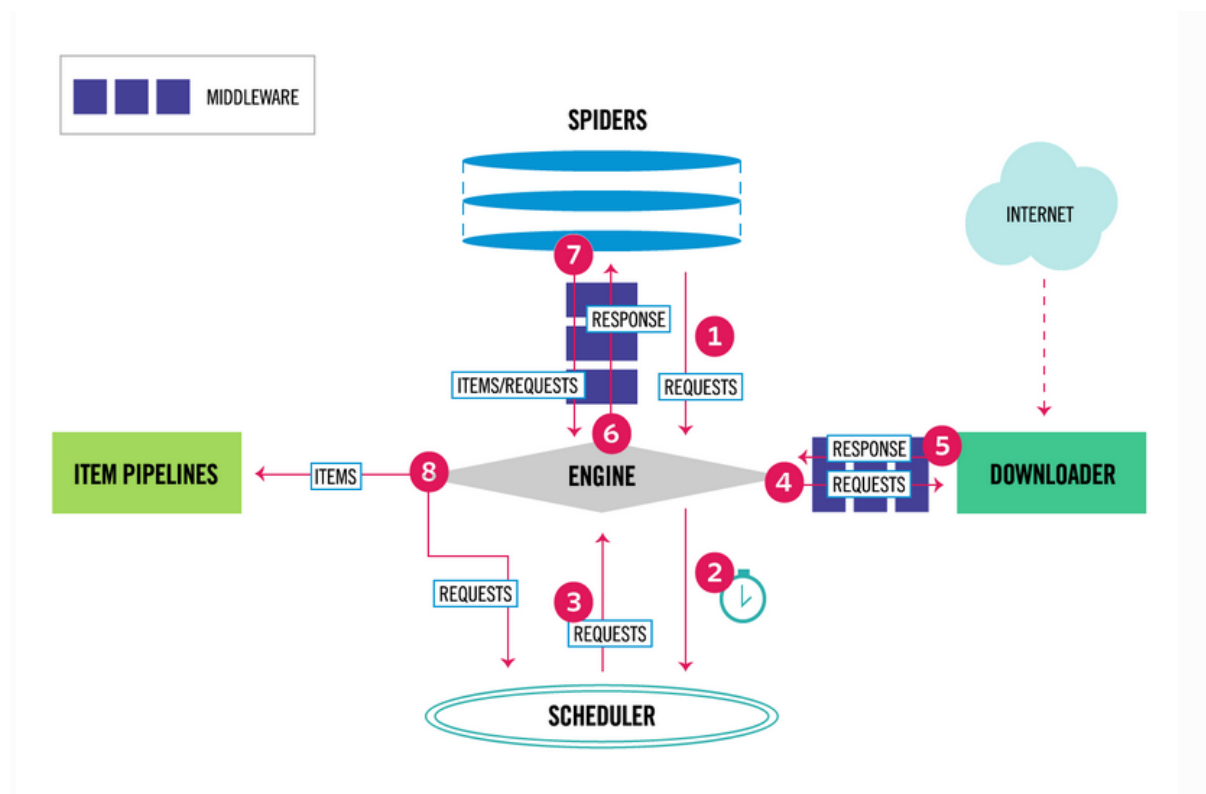


Figure 2.1: Scrapy Framework Architecture[1]

The general data flow in Scrapy occurs as follows

1. The Spider sends out a set of initial requests to the engine
2. The engine then schedules the requests in the Scheduler
3. The Scheduler then returns the next requests to the engine

4. The engine then passes the requests to Downloader through the Downloader middleware
5. The downloaded response is then sent to engine again passing through the Downloader middleware
6. The scraped information is sent by engine to spider in form of response.
7. The spider then processes the scraped data and returns the Items to engine
8. The items are then passed to Item Pipelines and if there are any more requests to be scraped they are sent to scheduler.
9. This process continues till the scheduler queue is empty.

2.2 FND System Schema

Using Scrapy allows us to use a common schema for the scraped data across all spiders. The schema is defined in *items.py* file. The common schema class (FNDFields in our case) inherits the *scrapy.Item* class. We define the fields as *scrapy.Field()* object. We define the following fields for our system.

1. **source_type:** The type of source can be classified into 3 major types for now i.e news_article, fact_checking and satire
2. **source_name:** The name of the site from which the article is scraped
3. **title:** The title of the news article scraped
4. **date_published:** The date at which the article was scraped
5. **image_link:** url of the images used in article.
6. **description:** A short description of the article if provided in article or first 5 sentences of article body.
7. **source_article_link:** URL of the article scraped
8. **article_body:** The text body of the article scraped
9. **time_scraped:** The time at which the article was scraped
10. **article_id:** Unique ID for the news article crawled. This is also the file name for the JSON file stored on disk.
11. **article_category:** The category of the news article like politics, sports etc extracted from the news source
12. **image_details:** The checksum and other related information of the image in article which is downloaded.

13. **debunked:** Denotes the credibility of the article based on source. We have currently 4 values for this field.
 - (a) *verify*: This information is available but the claim has to be verified by the user as true or not
 - (b) *fake*: The claim has been marked as fake in debunking sites
 - (c) *uncertain*: The system cannot label this article and expert opinion is needed to give judgement.
 - (d) *genuine*: The article is from a trusted source and is assumed to report the truth.
14. **ibm_nlu:** A higher level dictionary for all the data returned by IBM watson NLU API. This field contains additional information about the text body such as major entities, sentiment of the text, sentiments toward entities etc. This information allows us to do additional analysis and generate useful insights.
15. **tags:** The top entities and tags for the scraped article (extracted from source)
16. **ocr_english:** Contains English text extracted from images scraped from the article crawled
17. **ocr_hindi:** Contains Hindi text extracted from images scraped from the article crawled
18. **ocr_english:** Contains English text extracted from images scraped from the article crawled
19. **hash_value:** Contains a hash value calculated from images crawled. This is to speed up similarity search
20. **video_link:** Contains url of the videos linked in the article crawled.
21. **raw_html:** Contains raw HTML code scraped from URL. This is so that we have a way to extract additional information in future

2.3 Extracting data

To collect extensive data we need to extensively crawl the major Indian news media outlets. We choose a collection of Indian news sites from a mix of biased, fact-checking and genuine sites. This allows us to match information against news sources having different biases.

Every site has a different layout and the problem with deploying a single crawler for all sites using libraries like *goose* or *BeautifulSoup* is that, the data extracted is not clean. We deploy different crawlers for each website. Each crawler extracts information described in section 2.2. For this we take help of *Scrapy's response.css()* method. It uses the sites CSS layout to create a DOM model. We can then extract corresponding fields using the site specific queries. An example query is shown in below

```
#Extracts the description from website and stores it as description in dictionary
response_dict["description"]=response.css('span[class=mvp-post-excerpt\
left] p::text').extract_first()
```

We have deployed crawlers for the following sites,

1. Times of India

A trusted news media site owned by Bennett, Coleman & Co. Ltd. Its one of India's leading news publishers

2. The Hindu

One of India most popular English publishing houses, owned by The Hindu Group. Popular mostly in South India. We crawl this as a trust worthy news source.

3. Check4Spam

A Spam detection site which uses crowd sourcing to verify the truth of Whatsapp forwards and other rumours. Its operated by Shammash Oliyath and Bala Krishn Birla, operated as a non-commercial entity. We use this site as fact checking source.

4. BoomLive

Another fact checking site part of Ping Digital Network. This also depends on human expertise to classify claim as true or fake.

5. Firstpost

A news site part of Network18 media conglomerate owned by Reliance Industries. We classify news articles from this site as truthful.

6. Postcard

A biased news sites which provides a very biased and one sided articles. We classify all the news articles from this site as fake. This site started out as a facebook page and got subsequently banned from it. We do not include its results in the search results.

7. AFP Fact check

A fact checking site included by google in its Fact checking portal. We crawl it daily and also have crawled all the historical data.

8. AltNews

A Non-Profit entity which provides regular fact checks on recent political and other claims. We crawl it daily and also crawl archived articles.

9. India Today Fact check

A fact checking only version of the IndiaToday magazine. We crawl it daily as well all the archived articles.

10. News API

A Public API which indexes articles from over 30,000 worldwide sources. This aggregates news from various sources and provides a convenient access to multiple sources we are not crawling. We crawl this API endpoint every 3 hours

11. Quint Fact check

Fact check offering from the Quint magazine. One of the few fact checking sites from India approved from International fact checking network. We crawl this site every 3 hours.

12. Right Log

A right wing site which often has articles with a heavy right wing bias. We crawl articles from site and label them as biased.

2.4 Pipelines

To accomplish common task for crawled articles scrapy provides Item pipelines, they are generally used to do tasks like cleansing data, inserting it into databases etc. Each Item pipeline component is a python class which implements methods which are executed at different stages of execution. Code block 2.4 shows the basic structure of a pipeline and when the methods are called.

```
class PushSolrPipeline(object):
    def open_spider(self, spider):
        #Executed once at the start.
    def from_crawler(cls, crawler):
        #used to access core crawler objects like settings, signals etc.
    def process_item(self, item, spider):
        #executed for every article scraped, can be used to process data,
        insert into a database etc. Should return Item or Drop the item
        from pipeline.
    def close_spider(self, spider):
        #Executed once when the spider is closed.
```

We now take a look at every pipeline component in detail.

1. Validation Pipeline

This pipeline performs basic checks on the extracted data, ensuring that the data scraped is not blank. This pipeline mainly does the following things

- (a) Check if article body is not blank, if it is drop the item.
- (b) Check if the article is associated with a category, if not drop the item
- (c) Check if the article has an article ID , if yes check if it is a number and not text (as we use it as unique ID), if no drop the article.

2. **Stat Aggregation Pipeline**

This pipeline aggregates stats about the crawler run. Currently is a work in progress.

3. **WritetoFile Pipeline**

This dumps the crawled data dictionary in form of a JSON file. We store the JSON file (one per article) organized by the Crawler name and category of article.

4. **ImageInsert Pipeline**

This pipeline downloads a high resolution version of the linked images in the articles. This downloaded image is then indexed using the Image match library[7] into Elastic Search back-end. Indexing happens once per image. Along with the image the scraped data is also indexed by attaching it as meta-data. This helps us to get the source article directly when we query an image, instead of getting the ID and then querying SOLR.

5. **IndexSolr Pipeline**

This pipeline holds the extracted data in memory and Indexes all the articles crawled when the crawler completes its execution. The article is indexed using PySolr Library which communicates with the Solr REST API.

6. **NLU Pipeline**

We send the body of the scraped article to IBM Watson NLU API [3] to get different information like major entities in the text, sentiment of the article, sentiment towards entities etc. We store the response in `ibm_nlu` field of schema.

7. **OCR Pipeline**

Some of the images we crawl have text which are crucial for fact checking purposes. The information available in the image is sometimes available in the accompanying text but most of the times the text is not quoted verbatim. In such cases it becomes harder to cross reference and index the text available. We use Tesseract OCR engine [4] to do optical character recognition on the given image. The pipeline runs on the downloaded images and extracts both english and hindi text. This pipeline takes significant amount of time to complete and we run it only on daily crawls.

2.5 **Deployment**

Deploying and running the crawlers regularly can be done in different ways and each method has different pros and cons.

2.5.1 **Available options**

Previously the crawlers were run by executing them regularly through python scheduler which was called through a script which was running in background. The

problem with this approach is that the process might get killed and the crawlers may not run. Scheduling from scripts is always unreliable and may cause us to miss important news events.

Other option was to deploy the scrapers as daemon with the help of an inbuilt utility called scrapyd. This allows us to schedule spiders through a JSON API. The problem with this approach in the process of 'eggifying' the project all the paths and environment specific information is lost. This is problematic as we require this information to store data on Disk. The official documentation also advises against using scrapyd for projects which write to disk.

2.5.2 Approach taken

The option we used is writing cron jobs to schedule the spiders every 3 hours. Cron jobs are reliable and work as long the server is up. We use cron to execute a script which runs all the crawlers sequentially.

2.5.3 Running archive crawlers

Deploying archive crawlers comes with its own set of problems. The archives we crawl have almost 1 million articles which when scraping run for multiple days. If we schedule a cron job, is the process is killed the crawler stops functioning. What we needed was a process which starts itself after failure.

We created systemd services which restart on failure and runs the crawler 24/7 until completion. This method of deployment is suitable for large scale crawls which will work for extended period of time.

2.6 System services

In this section we discuss some of the system specific tools and architectures.

2.6.1 Directory structure

This section discusses the directory structure of the project. The scrapers-scrapy folder has all the scraping related code and the scraped data. The scrapy project layout is shown in figure 2.2. All the scrapers are in /scrapers-scrapy/fnd/fnd/spiders folder and all the other files like *settings.py*, *pipelines.py* etc are in the project base folder.

The *Data* folder in *scrapers-scrapy* contains all the scraped data. The directory structure is shown in 2.2. We maintain a separate folder for each crawler to organize the data better. Each scrapers folder has the following files

1. Category directories containing the JSON files of articles crawled. The categories are extracted from sites. Articles which don't have any category are stored in Misc. directory

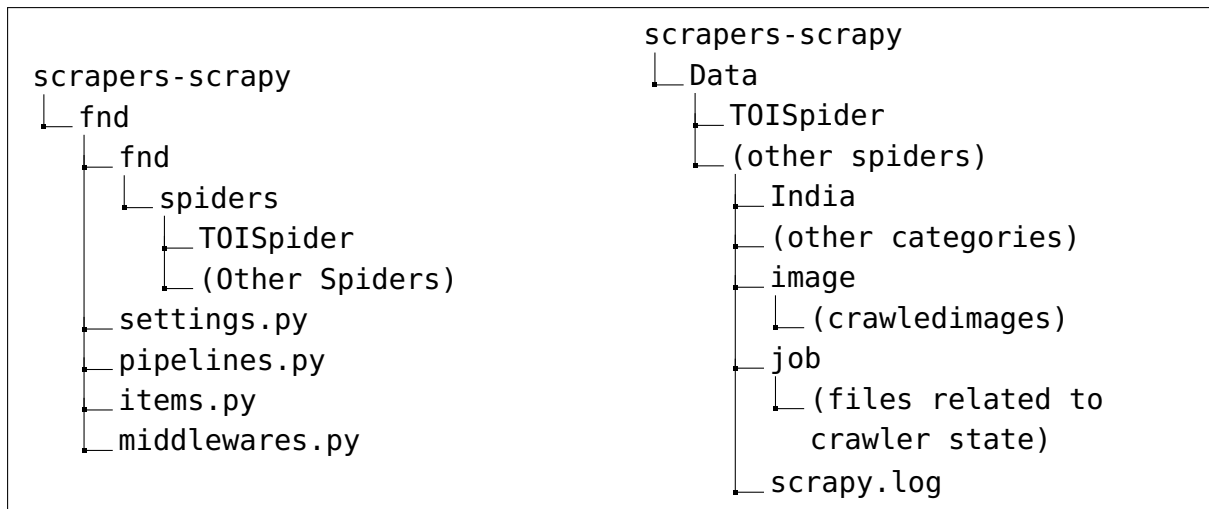


Figure 2.2: Scrapy Project directory structure

2. job directory contains all the book keeping information about the last spider run like requests seen, requests queue etc.
3. image directory contains all the images downloaded by the spider. They are stored as MD5 hashed file names.
4. scrapy.log file contains the logs from all the runs for the spider.

2.6.2 Maintaining the state of last crawl

While crawling it is important for the system to maintain the state of crawl till the last successful run. This helps us to do duplicate detection and also maintain stats of the run. Scrapy allows us to maintain state by using a Jobs directory which is specified by the *JOBSDIR* attribute in *settings.py*. The problem with this approach is doing this naively will create a common jobs directory across all the spiders. This is not desirable as this will cause conflicts between different crawlers and will also be highly inefficient. We solve this problem by specifying a unique jobs directory for each crawler, this directory follows the directory structure shown in ???. This is done by specifying the *JOBSDIR* parameter on a per crawler basis as is shown in code block 2.3.

2.6.3 Virtual Environments

There are number of different systems, tool-kits and APIs which work together to serve users request. Managing the dependencies becomes a major problem, also in a common environment these dependencies may interfere with each other causing problems. We use virtual environments to manage python based dependencies. Each virtual environment is stored separately and all the installed packages are stored in the virtual environment directory. The advantage of virtual environment is that the whole environment is wrapped in the python binary. So instead of activating the environment every time we have to run a command, we can directly run

```

class BoomLiveSpider(scrapy.Spider):
    name = "BoomLiveSpider"
    datadir = "/mnt/impecs/data/fakenews/scrapers-scrapy/data/"
    close_down = False
    language = "english"
    custom_settings = {
        'IMAGES_STORE' : datadir+name+'/image/',# Sets the image store for
            each spider
        'JOBDIR' : datadir+name+'/job/',# Sets Job Directory for each spider
        'LOG_FILE' : datadir+name+'/scrapy.log',# Specifies the log file for
            each spider
    }

```

Figure 2.3: Code to set jobs directory in spider

the command using the binary from virtual environment. For example running the crawler would be done efficiently done by the following command

```
/venv/bin/python3 scrapers scrapy
```

instead of

```
source /venv/bin/activate; python3 scrapers scrapy
```

This comes in very handy when we create systemd services. Using Virtual Environment also allows us to set proxy directly into binaries. This makes running crawlers 24x7 possible without any interruptions.

2.6.4 Logging

Considering we have many components interacting with each other we use extensive logging throughout our project. We log every stage of pipeline as well as crawlers. Logging this in a single file will make the file unwieldy and troubleshooting would be impossible. Instead we have separate log files for each crawler. We set them using `custom_setting` parameter in crawler as shown in fig 2.3.

2.6.5 Systemd services

Our system has multiple dependencies and the uninterrupted execution of these services is critical to the availability of the system. To ensure these systems start on boot and restart on failure we create user level systemd services for all the components of the system. These service's dependencies are structured in a hierarchical manner. We explain these services briefly below

1. `solrfdservice`

This service starts and ensures that the Apache solr[5] instance is always up and running. It is started on boot and has no dependencies.

2. `elasticfdservice`

This service starts the Elastic search instance[2], it restarts the service on failure. This service has no dependencies and starts on boot.

3. **imageapifndservice**

This service starts the Image search API server of the FND system . It starts after both elasticfnd.service and solrfnd.service start. This service also restarts on failure.

4. **primaryapifndservice**

This service starts the Fakenews API server. It starts after elasticfnd.service, solrfnd service and imageapifnd.service starts. This service also restarts on failure.

5. **Other services**

We have created other services for monitoring archive crawls, running other services like jupyterlab etc.

2.6.6 Cron jobs

The crawlers scrape the websites listed in section 3.3 for latest news articles. The crawlers run every 3 hours every day. We have cron jobs to crawl the sources by the frequency mentioned.

2.7 Challenges and their solutions

There were some challenges in scraping news articles and we discuss them and the workarounds we have applied in this section.

1. **Binding the crawl**

Scrapy by default crawls in a Depth First order, which would be a problem for us as we need to do only a 1 level traversal in most cases and 2 level in worst case. So travelling in Breadth first order is required to bind the crawler. Scrapy allows doing that by setting the following parameters in *settings.py*

```
DEPTH_PRIORITY = 1
SCHEDULER_DISK_QUEUE = 'scrapy.squeues.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeues.FifoMemoryQueue'
```

2. **User Agent blocking**

Scrapy by default uses Scrapy/VERSION(+https://scrapy.org) as user agent. Many publishing sites block unknown user agents and only allow user agents corresponding to browsers to access content. If you use any other user agent the site will give a *403:Access Forbidden* error. So we set the user agent in *settings.py* as follows,

```
USER_AGENT = 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0'
```

This corresponds to Firefox browser.

3. **Crawling Archives**

We decided to crawl the archives of major news sites like Hindu and Times Of India to have a comprehensive historical data set of news articles. Some sites deploy some protections to make the archives crawler unfriendly, for example TOI calculates the no. of days from some random day in 1800s and uses it in URL along with the month,day and year of the archive list. Figuring this out required analysis of all the JavaScript files and figuring out the logic so we can crawl directly.

4. **Downloading Images**

Sites such as Hindu display images depending on the resolution of the display used to access the image. Crawling them directly would fetch the lowest resolution image which would be problematic as we lose information when we reduce the resolution. This may prove to be problem when the no of images indexed increase, this may possibly give us false positives when queried. We dynamically generate the image URL so that we get the highest resolution image.

5. **Handling Duplicates**

Handling duplicate URLs is a very important part of the crawler, initially we designed a Bloom-Filter based duplicate detection mechanism which worked well but had some drawbacks and holding it in memory was a challenge. We are currently using scrapy's *RFPDupeFilter* class, which filters requests based on their fingerprints. We plan to write a custom class in future which utilizes a lightweight database but since the current duplicate filter works reliably we are treating this change as an optimization.

6. **Indexing Services unavailable**

Indexing information into Elastic Search or Solr is just a single step in pipeline and as such the article is saved in a JSON file even if it is not indexed. This poses a problem as we are not keeping track of articles crawled and saved but no indexed. To avoid this, we shut down the spider if the services are unavailable. Closing the crawler in scrapy cannot be done from pipelines, so we instead set a flag from pipeline which forces the crawler to shut down. We can afford to do this as our repetitive crawls are generally crawling the same content in multiple iterations.

7. **When to index in SOLR?**

SOLR allows you to index data through its REST API, the problem is this involves sending an HTTP request. Though we don't have to transmit data over network (as crawler and solr run on the same system), the overhead doing this once for every article scraped adds up and is undesirable. Instead of indexing every article separately, we hold the crawled articles in memory and index all the crawled articles at end. This has a possible downside of loosing on durability property if the crawler abruptly exits. But considering

our crawlers work every 3 hours the data lost will be re-indexed in the next crawl.

8. **Index images into elastic search**

The first iteration of the Fakenews API enabled both indexing and querying images over a REST API. But this arrangement posed a potential threat, what if some adversary decides to push images which propagate the fake news instead of original one? This means that the user will see the malicious image instead of original one. Initially the ImageInsertPipeline would index images into Elastic search back end using the API, but since we have closed that API endpoint we directly index into Elastic search using the Image-match[7] library. This speeds up the pipeline and reduces unnecessary hops to travel.

2.8 Improvements and tweaks

1. **Exclude Tag field for searching documents:**

Tag field contains important entities in the article. This throws up irrelevant articles when included in search fields. So we search without the Tag field given a text article and only if that query returns no results we include the field in query field. This reduces the number of irrelevant results considerably.

2. **Log failed queries:**

Queries which fail in our system need to be logged so that we might get them fact checked later and inform the user. We log the failed queries in a postgres database named 'missed_queries'. We store the time of the query, the query text and the image url and link to article if present. We also store the email ID of the user querying to return results in the future if the query is resolved.

3. **Remove duplicate results:**

Multiple crawls or same crawlers run multiple times to test some configuration change used to throw up duplicate entries. We used the group_by setting in Solr to remove these duplicates. This change required a major rewrite of the front end code. The results are now grouped by URLs and hence there are no duplicates. Some cases like same article referred by slightly different URLs do throw up duplicates but such cases are very few and can be ignored.

4. **Changing the query parser:**

To enable advanced search and filtering options we switch the query parser from the DisMax query parser to eDisMax query parser which allows us to use advanced search features like using conditional operators such as AND, OR etc and support for full lucene syntax.

5. **Change all the requests to POST:**

Fake news API initially worked by sending GET requests to the API endpoint. Given the security considerations and the possibility of exposing users Identity through the queries sent we switched to a POST based model for the front end API.

6. Upload size max limit:

Nginx limits the size of maximum uploads to using a configuration setting. We had to change this setting in the host web sever as well on the reverse proxy to enable large uploads for videos.

7. Limit search to title:

Small queries throw up irrelevant results when searched normally in solr. We instead search only on title field if the given query is less than 5 words in length. We search using our initial setup if the length of query is greater than 5.

8. Boost recent articles:

The results in solr are ordered by the score in default configuration. We want the recently crawled articles to be on top. We set the boost field in solr to boost scores of articles which are published closer to the current date. We set this parameter while querying. Since this involves using the date_published field which was initially configured as a string in Solr, we had to re-index the whole core again.

Chapter 3

Political Bias Detection

The previous part of the report concentrated on acquiring data and searching it efficiently to see if similar articles have been debunked or reported on previously. In this section we use various techniques to identify and detect political bias in a given text. There are multiple approaches and we study them in detail in the following sections.

The past approaches to detecting political bias work by extracting linguistic features from text and then using machine learning models like SVM, decision tree etc to predict the category which is a generally a variation of how fake or biased the given text is. Some datasets like semeval dataset [6] label the text based on source(or publisher in this case) and others label individual articles by human fact checkers. Both of these approaches have problems, if we use human fact checkers the amount of data we get won't be enough to train a good model. This can be seen in the semeval dataset where the per article labelled dataset only has 645 articles. The other approach is labelling the text by source. The semeval dataset does this by using as source of bias of a news source. There are multiple problems with this approach

1. The credibility of labelling source is a major factor. People generally tend to dismiss sources which conflict with their view point. Convincing people about the bias of a particular source or article becomes easier if there is a mathematical way of explaining the bias labels.
2. There is an implicit assumption that all articles from source have the same bias as the sites but we know that this isn't true. Even biased sites publish articles which are credible, labelling them all as biased would lead to loss of credibility. We need to generate bias labels on a per article basis
3. One more problem with annotating based on human fact checkers is gauging political bias is a very subjective problem. Articles which are judged to be biased towards one political entity by one factchecker may be shown to be biased towards other entity by other author. This subjective bias in labelling articles by authors can be mitigated by increasing the number of fact checkers labelling the same article and taking a majority vote on the label for the article. This additional overhead along the with the cognitively taxing job of

reading an article and finding its bias makes annotation of news articles by human factcheckers unviable.

3.1 Previous Work : Hyper Partisanship detection

Hyper partisanship is defined as "exhibiting blind prejudiced or unreasoning allegiance to one party, cause or person" [6]. Majority of fake news in social media circles can be seen to come from relatively unknown media sources with dubious credentials. This content is heavily subjective and in general is worded differently. The prevalence of such sites on known 'troll' pages is a matter of concern as many people mistake them for actual media houses. These pages also name the domains similar to established houses, this causes confusion among the users and they take the information disseminated on them on face value.

Hyper partisan articles are easy to identify by humans but doing automatically is hard. We plan to identify the bias in articles towards right wing or left wing. Semeval 2019 task 4 tackles this problem and provides a comprehensive dataset of a million labeled articles to test our models. We discuss it in detail in the next section.

3.1.1 Semeval 2019 Task 4: Hyper partisan News detection

Hyper partisan News detection is one of the tasks in Semeval 2019 workshop on Semantic evaluation. Following are the details about the task taken from the official website [6]

1. Task

Given a news article text we wish to classify the partisanship of the text in one of the five categories i.e right, right-center, least, left-center, left.

2. Data

The organizers provide 1 million articles labelled by hyper partisanship and bias. Hyper partisanship is a binary label, whereas bias can be one of five classes right, right-center, least, left-center, left. The dataset is split 80-20 into training and validation sets.

Modelling this as a multi class text classification problem with the given text data as input and one of five classes as output we managed to achieve a categorical accuracy of 90% on test set. The accuracy is the mean accuracy rate across all predictions for multi class classification problems. The model uses TFIDF vector representation over a vocabulary of 30000 words. These are then supplied to a neural network with two dense layers of 512 units each and final dense layer of 6 units to generate labels. This neural network surprisingly performs well given its simplicity. The model has 15626246 trainable parameters which are trained through Adam optimizer. We plan to improve this model by using other optimization and adding linguistic features.

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 512)	15360512
activation_7 (Activation)	(None, 512)	0
dropout_5 (Dropout)	(None, 512)	0
dense_8 (Dense)	(None, 512)	262656
activation_8 (Activation)	(None, 512)	0
dropout_6 (Dropout)	(None, 512)	0
dense_9 (Dense)	(None, 6)	3078
activation_9 (Activation)	(None, 6)	0
Total params: 15,626,246 Trainable params: 15,626,246 Non-trainable params: 0		

Figure 3.1: Neural network Architecture

3.2 Political Bias detection using text in Indian Media

Detecting political bias is useful in Indian Context as the current political climate in India causes people to distrust media sources which do not adhere to their biases. People are more likely to question the credibility of the news source rather than government policies, one of the best indicators of this is the comments section of major media sources on social networking sites.

Many media houses are clearly politically biased. These biases can be attributed to the ownership patterns of media conglomerates, the political alignment of the owners or the prevailing political inclination of majority. Our aim is to detect this political bias and give users an idea of the bias a certain publisher has.

- **Limitations**

Our initial plan was to create a dataset similar to the dataset in the semeval task but suited to Indian context. There were some limitations in this approach which we discuss briefly

1. We lacked access to human annotators which the semeval data set used to manually label articles as hyper partisan or not.
2. Labelling by source is problematic and the assumption that all articles from a particular source are biased towards some political ideology doesn't hold up in real world. Many sites regularly publish articles which are trustworthy but some articles are particularly biased.

3. Source bias changes according to various factors like ownership of publishing house, current political climate, public opinion etc.

To overcome these limitations we change our problem statement to Identifying political bias of a given text instead of hyper partisanship detection. Given an article we wish to classify the article as biased towards one of the multiple political parties and their allies. We describe in detail the methodology we applied to get the clusters and how we train different models to predict the bias

3.3 Methodology for crawling twitter

To label the crawled data we take signals from social media more specifically from twitter.com . We use twitter to identify community of users who show a bias towards a particular political entity (in our case the three major political parties and its allies). We crawl twitter using an initial list of 250 IDs scraped from a social media analysis site which lists political handles called and additional 400 IDs scraped from official pages of political parties. These ID act as initial set of IDs for which we crawl all tweets which mention some other user. We filter out the tweets which have mentions in them by using twitter advanced search features. We get the store the tweet id, text, retweet count and favourite count. Once we finish this is initial crawl we select handles for the next level of crawl using the following heuristic

1. Get all the twitter handles mentioned in the tweets
2. Aggregate the count of retweets for a particular user over all tweets
3. Filter the handles by selecting only handles above a certain threshold (100 in our case)

These handles then act as seeds for the next crawl. We can then repeat the process multiple times to obtain deeper crawls. We do multiple levels of crawls to get roughly 676000 tweets. This consists of tweets scraped from 6473 handles.

3.4 Constructing Influence Graph

Next we construct a graph with information collected in the crawls. We have two ways of constructing the influence graph. We initially started with using user mentions as a form of endorsement but Conover et al. [8] came to the conclusion that using retweets as endorsements leads to better clustering. The paper by Conover et al. comes to following conclusions

1. Retweet Network is highly modular and easily separates into right wing and left wing clusters as opposed to the mention cluster which performs very badly.
2. Retweets are endorsements, people retweet those who they agree with politically. Mentions form a bridge between different ideologies.

3. Users interact with people in the community using retweets and between others in opposing community using mentions.

So we use two different approaches to construct influence graph. First we construct a graph using user mentions and then we use the same method to construct a graph out of retweets.

3.4.1 Mention graph

We construct a directed graph G by using the heuristic mentioned below.

1. The nodes here (V) are twitter handles, since the handles are unique we convert all of them to lower case to avoid false mismatches due to case changes.
2. The directed edge from point a to b exists if a tweet from user a mentions user b .
3. For the weights of the edge we use 3 different methods
 - (a) The edge weight is the number of times a mentions b
 - (b) Instead of treating each mention similarly we use sentiment analysis to analyze the sentiment of the tweet, we add the sentiment score (between -1 to 1) of the tweet every time a user a mentions b . We shift the scale of sentiment between 0 to 2 by adding 1 to sentiment score to remove negative and 0 weighted edges.
 - (c) Some tweets are more influential than others. For example, if there are 100 tweets with positive sentiment between two users and only 5 negative tweets we shouldn't be treating the sentiment scores of both similarly. Instead we multiply the sentiment score of a tweet by the number of retweets it got. We then add this score every time user a mentions b .

3.4.2 Retweet graph

We construct a similar directed graph G for retweets using a slightly different heuristic mentioned below mentioned below.

1. The nodes here (V) are again twitter handles which we convert to lower case.
2. The directed edge from point a to b exists if a tweet from user a retweets a tweet from user b .
3. Since retweets do not contain accompanying text we set edge weights as the number of times user a retweets user b .

After we construct the graph we use clustering algorithm by Blondel et al. [9] to find out the communities in the graph. We discuss this algorithm in detail in section 3.4.3

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j),$$

Figure 3.2: Modularity

3.4.3 Clustering Algorithm

The clustering algorithm by Blondel et al. [9] works by maximizing the modularity measure as defined by Newman et al [10] . The algorithm initializes all nodes as separate clusters and merges neighbouring nodes in single cluster such that the modularity is maximised. The algorithm converges when modularity doesn't increase anymore. We use a variant of this algorithm for our task. The modularity formulation by Newman et al. [10] measures the density of links inside the communities as compared to the the links between communities and is given by 3.2

Here A_{ij} represents the weight of the edge between nodes i and j . Also $k_i = \sum_j A_{ij}$ represents sum of all the weights of edges attached to vertex i . Community of vertex i is given by c_i , the function $\delta(c_i, c_j)$ is 1 if $c_i = c_j$ and 0 if not. m is a scaling factor which is given by $m = 1/2 \sum_{ij} A_{ij}$

We cluster the graph recursively, initial clustering gives us handles which belong to various areas like politics, bollywood, sports etc. We then keep only the handles in the political cluster and remove the rest. Now we apply clustering algorithm once again to further refine the clustering we got. After this clustering operation we get clusters corresponding to the major political parties in India i.e INC and BJP. Other political parties and their allies are in a separate clusters.

Our approach to clustering using mentions takes the sentiment of tweets into account while constructing the graph in form of weights. This approach helps us take into account the linguistic features into account while constructing the graph as opposed to only the connectivity/endorsement features. Ozer et al. [11] try to do the same by incorporating a user-word matrix which helps capture words being used by a particular user. We instead incorporate sentiment information directly into the graph structure. This helps us generate better clustering using the same algorithm by Blondel et al [9].

The clustering algorithm can be controlled using a resolution parameter. The default value of this parameter is set to 1. Setting it higher reduces the number of communities and setting it lower than 1 increases the number of communities. We experimented with different values o We initialize the algorithm with different modularities to find out which provides best clustering.

One of the main reasons we use such a simple formulation instead of more advanced and task specific formulations is the scalability of the algorithm. This algorithm scales easily to more than a million nodes. Other algorithms don't work as effectively on large networks like we have.

3.5 Clustering Results

We run the clustering algorithm on both graphs. Fig 3.3 shows the clustering we get on the mentions graph. The clusters we get correspond to various different entities. Analyzing the clusters we can see that cluster we get can be grouped according various parameters into groups as

- Handles related to Bhartiya Janata party
- Handles related to Indian National Congress
- Handles related to but not limited to Aam aadmi party
- Publishing houses and their twitter handles
- Bollywood and Entertainment Industry
- Handles from the USA

We similarly cluster the retweet graphs and we get similar results. The results were a bit clearer and the handles corresponding to political handles were correctly classified. We use the retweet graph in our final results.

3.6 Organizing and Crawling Data

The approaches we take depend heavily on the data we scrape from twitter.com. In this section we discuss the methodology we follow to scrape data. We discuss the crawling methodology and how we organize this crawl.

3.6.1 Tweets

Gathering data from twitter is one of the most important components of our system. To ensure that the clustering is good enough we need comprehensive crawls which focus on relevant twitter handles only. We use two approaches to crawl twitter

1. Crawling twitter through official API imposes limits on access which are refreshed after sometime. Initially we crawled twitter using Scrapy framework [1]. Which functioned by scraping the generated HTML page after searching through twitter search feature. The information scraped using this method is not as rich as the one we get after using official twitter API, we mainly miss out on retweet information. Also one of the major problems we have is the links scraped from tweets frequently have spaces in URL which are very hard to correct. Instead of crawling all the tweets we only crawl tweets which mention some user in the first phase and in the second phase we crawl only tweets containing some links to articles. In total we crawl ≈ 900 thousand tweets using this method

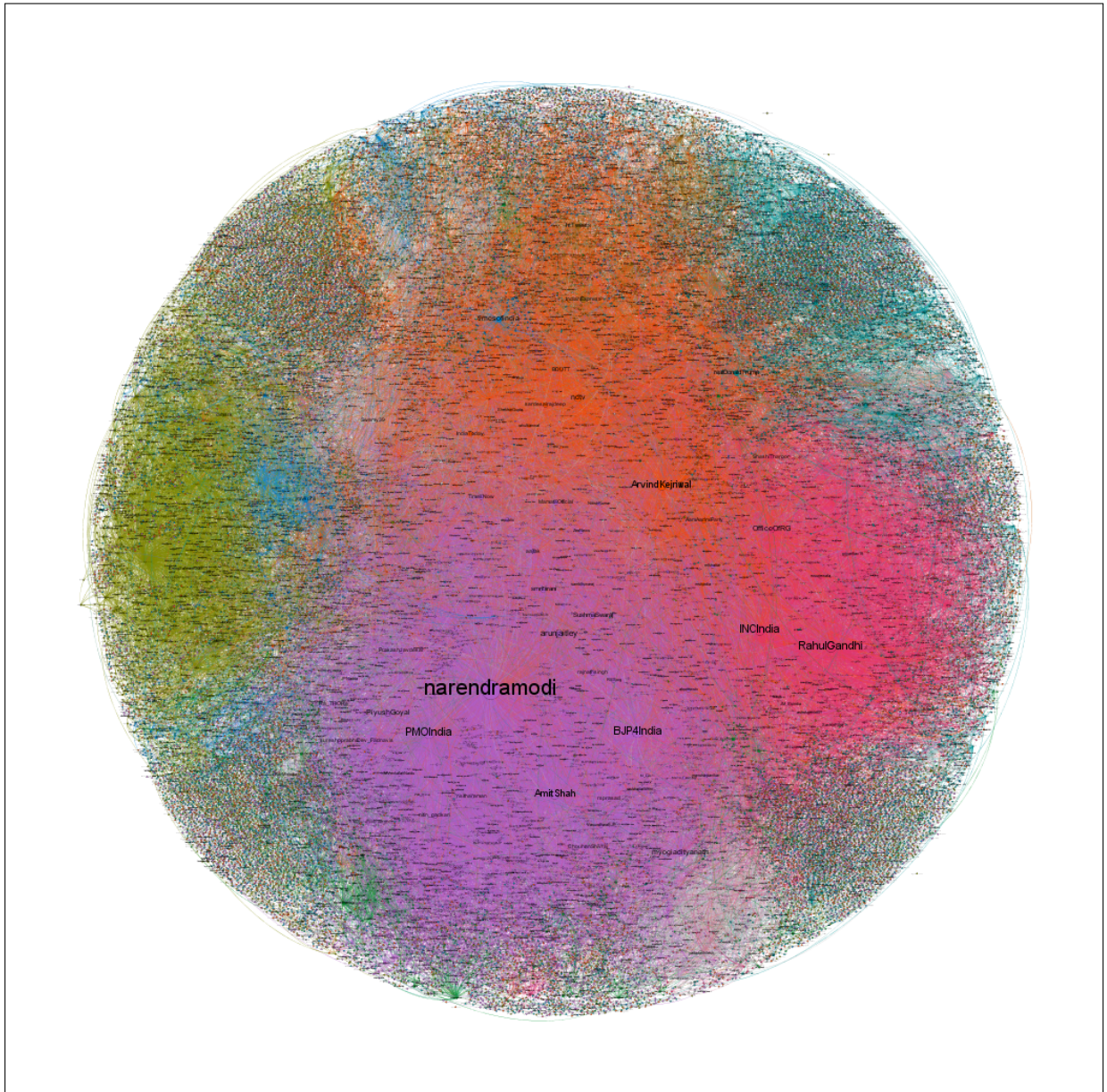


Figure 3.3: Clustering results

2. Official Twitter API allows creation of multiple apps, which allows us to crawl at a significantly faster speed. We use tweepy library as a wrapper over official twitter API and write multi-threaded code to crawl tweets from a set of users. This code works reasonably fast and provides us richer information. Here each thread works with OAuth tokens corresponding to one application. We wait if the API limit is exhausted till the quota is replenished. Using this method we crawl ≈ 40 Million tweets.

The data scraped from twitter is stored in a PostgreSQL database to ensure high availability and concurrent access to multiple users. We store data scraped from both the methods in separate tables with Primary key constraint on Tweet ID. We construct an Index on Tweet ID column for faster access.

3.6.2 Cluster IDs

Once we have the tweets we construct graph and find out the cluster IDs of every handle as explained in 3.4 and store the information in a table. This for finding out cluster labels of tweets and articles easily using simple SQL join queries.

3.6.3 News Articles from twitter handles

Once we have twitter handles and the communities we select handles which belong to specific clusters which we are interested in. We crawl the URLs to news articles from these handles. Since the domains can be different we use scrapy framework along with Goose library to extract text and other data from the given URL. This allows us to write a one general purpose scraper which will crawl all the URLs. Scrapy's working is explained in detail in Chapter 2.1. We have two Pipelines for this crawler

1. **WriteToFile Pipeline:** We store the crawled tweet in JSON file.
2. **InsertIntoDB Pipeline:** We insert the crawled data into PostgreSQL database.

3.7 Text Classification

Once we gather data we have data in the form of Text and their labels. These labels correspond to the cluster IDs to which the user who tweeted the article belonged. We then use various neural architectures to do a 3-label classification task. We discuss some of the approaches we take and their results

3.7.1 Universal Language Model Fine Tuning

We use this model architecture by Ruder et al. [12] to predict cluster IDs given the text. ULMFiT works on the principle of transfer learning where we train a part of the neural network (encoder in this case) on a general purpose task and then use this pre-trained network as a component in the final model. Then we fine tune

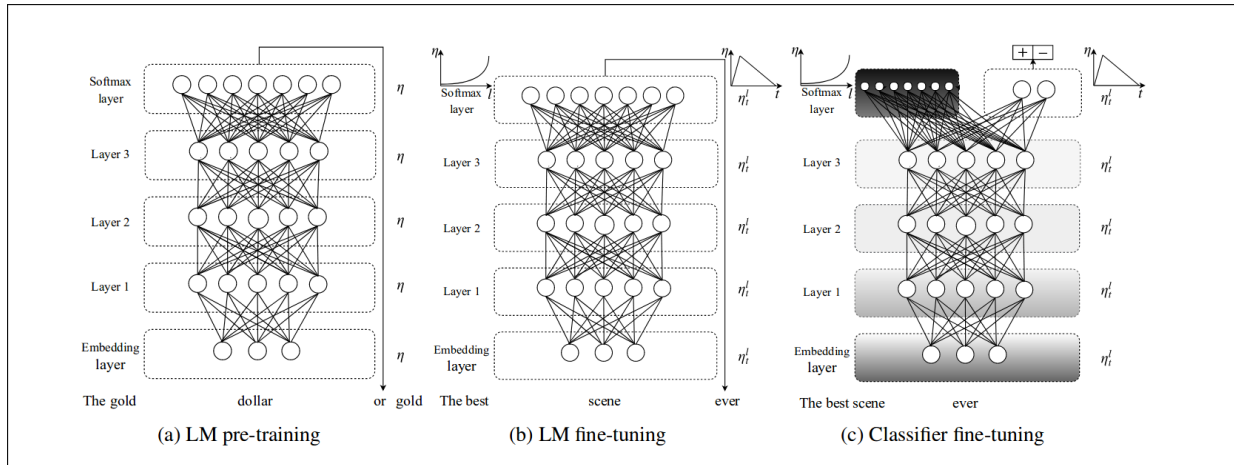


Figure 3.4: ULMFiT model Architecture [12]

this pre-trained model for the classification or other NLP task .One simple way transfer learning was used in NLP before was in the form of word embeddings like Word2Vec or GloVe. This technique only targets a models first layer as opposed to ULMFiT which affects more layers.

Ruder et al. argue that language model training is the perfect task for transfer learning in case of NLP. Language modeling captures many linguistic features like long-term dependencies, hierarchical relations, sentiment etc. which are useful for tasks in NLP like classification, question answering etc.

The working of ULMFiT model can be summarized in Diagram 3.4

3.7.2 Our Setup

We use the Pretrained Language model trained on Wikitext-103 dataset which consists on 28595 Wikipedia articles. We fine tune this language model to our task using the dataset we have from crawling Indian news sites as detailed in 2.1. Initially we used all 2 Million articles to fine tune the Language model but according to the author training the language model above 1 million tokens causes no significant gains. So we only fine tune the language model only on political articles. We select articles based on the article categories given by the publishing houses. Once we fine tune the language model we train the classifier on the dataset which consists of article body as input and cluster IDs as labels.

We organize the experiments in two main categories

1. Classification using article body:

We use the article body as the input to the neural network the labels are the cluster IDs

2. Classification using article title:

The title of article contains maximum information about the article according to literature related to text summary. We do classification using title of articles as input and cluster IDs are labels.

The data we have is summarized in table 3.1. We keep only the articles from cluster IDs 0,3 and 10. These handles roughly correspond to the political twitter handles from Bhartiya Janata party, Indian national Congress and Aam aadmi party respectively. We can see the that from the most important handles in the clusters as shown in fig 3.2

Cluster	No. of samples
Cluster 0	1221499
Cluster 3	516926
Cluster 10	25326

Table 3.1: Data

We can see from table 3.1 that the data is heavily imbalanced. We use both upsampling and downsampling and report results on both data sets.

Cluster 10	Cluster 3	Cluster 0
cnnnews18	ani	indiatoday
arvindkejriwal	narendramodi	rahulgandhi
maryashakil	timesnow	shashitharoor
drkumarvishwas	timesofindia	ram_guha
ashu3page	pmoindia	svaradarajan
ashutosh83b	sushmaswaraj	incindia
vishaldadlani	shivarooor	priyankac19

Table 3.2: Top twitter handles from each cluster

3.7.3 Results & Discussions

We report a 78.4% validation accuracy on the up sampled dataset whereas the downsampled dataset gives us a maximum accuracy of 59%. We can see the confusion matrix from upsampled dataset in fig. 3.5 and from the downsampled dataset in fig 3.6

The results we get are not satisfactory and linguistically there doesn't seem to be much difference in the articles between clusters in our dataset. This might be explained by the fact that both the major parties in India tend to talk about same topics. To confirm this, we ran took the intersection of top 100 entities from both clusters and found out that 77% of them talked about the same entities.

This confirms our suspicion that linguistic features only are not enough for detection of political bias.

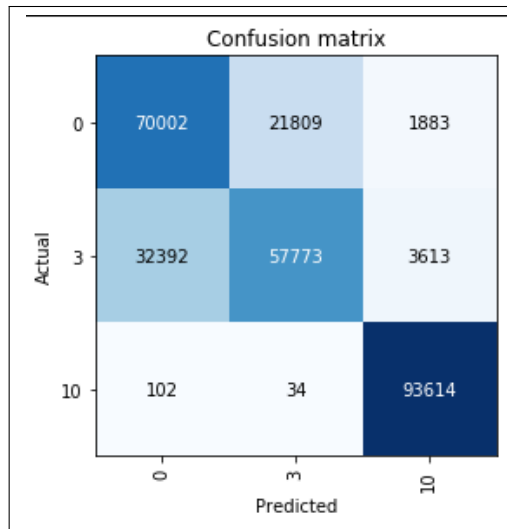


Figure 3.5: Confusion matrix for Upsampled dataset

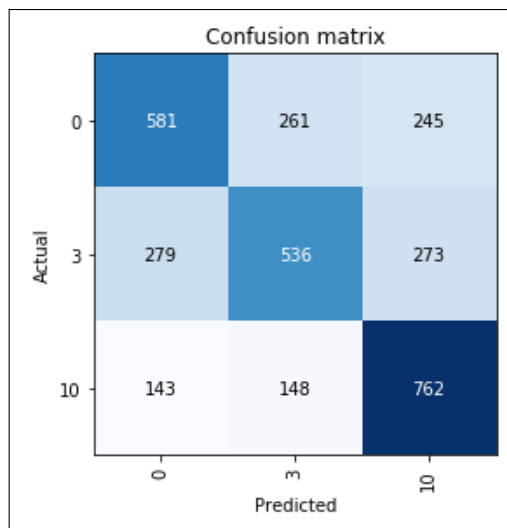


Figure 3.6: Confusion matrix for Downsampled dataset

Cluster 10 (AAP)	Cluster 0 (INC)	Cluster 3 (BJP)
('toi.in', 0.35987261146496813)	('bhopale', 0.03225204491027276)	('sanskritimagazine', 0.09522929848058005)
('tweeetdntimes.com', 0.1316348195329087)	('dlvr.it', 0.029484029484029485)	('rupavahini', 0.07707938853738512)
('satyahindi', 0.0329087048083227176)	('bbmp.sahaaya.in', 0.029204117811712747)	('nationmultimedia', 0.043966194060869164)
('indianexpress.com', 0.01910828025477707)	('patrikai.com', 0.027462445183964172)	('nm4.in', 0.035607075232069456)
('tl.gd', 0.014861995753715499)	('lnkd.in', 0.025876279040836003)	('nethratv', 0.024892624578580335)
('timesofindia.indiatimes.com', 0.013800424628450107)	('buff.ly', 0.021988616925325787)	('appritam.com', 0.023276226385166028)
('app.nbt.in', 0.012738853503184714)	('aksharnama', 0.01794544832519516)	('timesofindia.onelink.me', 0.017872812081466773)
('me', 0.010615711252653927)	('sudditv', 0.016452586072839236)	('pgportal.gov.in', 0.013393063316861405)
('firstpost', 0.009554140127388535)	('hastakshep', 0.01592386402512985)	('lnkd.in', 0.01283886759340507)
('ketto', 0.009554140127388535)	('nytimes', 0.015332939383572296)	('ns7', 0.011545744238673624)
('ourdemocracy', 0.009554140127388535)	('sbp', 0.01418219139738127)	('ndtv', 0.010806816607398514)
('org', 0.008492569002123142)	('congress', 0.01399558361583678)	('flipa.com', 0.010760633630443819)
('forbesindia', 0.008492569002123142)	('timesofindia.onelink.me', 0.011694087643454733)	('commun.it', 0.009098046460074817)
('ndtv', 0.0074309978768577496)	('nyti.ms', 0.011538581158834323)	('epfigs.gov.in', 0.008174386920980926)
('barandbench.com', 0.006369426751592357)	('deshabhimani', 0.010667744844960034)	('dhunt.in', 0.007943472036207454)
('hindustantimes', 0.006369426751592357)	('indianexpress.com', 0.010543339657263708)	('cmun.it', 0.007204544404932342)
('amp.twimg.com', 0.006369426751592357)	('bansalnews', 0.010170124094174728)	('timesofindia.indiatimes.com', 0.006881263566249481)
('forbesindia.com', 0.006369426751592357)	('amzn.to', 0.009641402046465337)	('bhaskar', 0.006881263566249481)
('nia', 0.006369426751592357)	('thehindu', 0.008210742387957578)	('8ththeatreolympics', 0.006604165704521313)
('thehindu', 0.005307855626326964)	('bloomberg', 0.008086337200261251)	('timesofindia', 0.006234701888883758)

Figure 3.7: Source Based Political Bias detection

3.8 Source Based political bias detection

We try another approach to find out bias. We use the clusters we get in section 3.5 to perform a source based analysis similar to the one in [6]. We use the clusters we get to get the source bias of a particular source. Source based bias works pretty well for extremely biased sites. These sites consistently push out narratives which benefit a particular entity. The tricky part is identifying the political bias of Major media houses. These sites generally show a subtle bias which would be hard to pick up from linguistic signals alone, source based political bias detection does a better job in this case. The methodology we followed is as follows

- We gather twitter mentions from twitter using crawlers as detailed in 3.3
- We then construct the Graph using either mentions or retweets similar to what we do in section 3.4
- We run the clustering algorithm described in 3.4.3 to get clusters
- Then we get all the URLs mentioned in tweets from every cluster.
- We extract the domain names for every cluster.
- We compute the proportion of articles from a particular source in a particular cluster
- We categorize the source bias towards political entities in the cluster where the proportion of the sources is highest

We find out this corresponds well with the generally held beliefs about source biases and shows improvements over the previous approach. After filtering out the irrelevant sites like facebook, link shortening sites etc. we can see that the clusters do share biased news articles more. We display the results from this exercise in figure 3.7 The overall methodology is simple and works in most of the cases. One limitation we found was that source bias calculated in such a way changes over time. This makes calculation of the bias slightly harder. We leave this problem to be explored further.

Chapter 4

Summary and Conclusions

In the first phase of the project we developed a system to crawl data and index it with Apache Solr for searching effectively against the information already available on fact checking and other news sites. We describe the challenges and problems faced and how we solved them in this report.

In the second phase of the report we do various improvements to the system and explore Political bias detection. We take the help of Twitter to find out bias of users using Influence graph and construct a dataset with article body and the bias they show towards a political cluster. We train the current state of the art classifier on it to train a neural network which will identify the political bias of a given text. After seeing the results we come to the conclusion that using only text to find out the political bias is hard because of imperfect clustering and limitations of using only English text.

Bibliography

- [1] ScrapingHub, “Scrapy: A framework for extracting the data you need from websites.” 2018. [Online]. Available: <https://scrapy.org/>
- [2] ElasticsearchBV, “Elasticsearch is a distributed, restful search and analytics engine.” 2018. [Online]. Available: <https://www.elastic.co/products/elasticsearch>
- [3] I. corporation, “Natural language processing for advanced text analysis.” 2018. [Online]. Available: <https://www.ibm.com/watson/services/natural-language-understanding/>
- [4] R. Smith, “An overview of the tesseract ocr engine,” in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, ser. ICDAR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 629–633. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1304596.1304846>
- [5] ApacheSoftwareFoundation, “Solr is the popular, blazing-fast, open source enterprise search platform built on apache lucene,” 2018. [Online]. Available: <http://lucene.apache.org/solr/>
- [6] Semeval2019, “Semeval task 4 : Hyperpartisan news detection,” 2019. [Online]. Available: <https://pan.webis.de/semeval19/semeval19-web/>
- [7] ascribe, “Image match library,” 2018. [Online]. Available: <https://github.com/ascribe/image-match>
- [8] M. D. Conover, J. Ratkiewicz, M. Francisco, B. Goncalves, A. Flammini, and F. Menczer, “Political Polarization on Twitter,” p. 8.
- [9] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct. 2008, arXiv: 0803.0476. [Online]. Available: <http://arxiv.org/abs/0803.0476>
- [10] M. E. J. Newman, “Analysis of weighted networks,” *Phys. Rev. E*, vol. 70, p. 056131, Nov 2004. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevE.70.056131>

- [11] M. Ozer, N. Kim, and H. Davulcu, "Community detection in political twitter networks using nonnegative matrix factorization methods," *CoRR*, vol. abs/1608.01771, 2016. [Online]. Available: <http://arxiv.org/abs/1608.01771>
- [12] J. Howard and S. Ruder, "Fine-tuned language models for text classification," *CoRR*, vol. abs/1801.06146, 2018. [Online]. Available: <http://arxiv.org/abs/1801.06146>