DeepSearch Labs

Task Report

Author: Vishal Kashyap

List of content

List of files

/BBC_articles

bbc_df.csv :	Scraped dataframe
bbc_scraper.py:	Script for scraping website
flask_app.py	Script for web application
/images	Stores all scraped images
/templates	Files for flask web application

/Knowledge_graph

Knowledge_graph.ipynb:	jupyter notebook for constructing knowledge graph
kg_coref.csv:	dataframe of extracted relations using algorithm described in report
kg_spacy.csv	dataframe from anther algorithm tested (no coreference resolution)
kg_flair.csv	dataframe from flair's relation extractor. (limited relations)

1. Knowledge graph:

A knowledge graph comprises entities and relations. The nodes of the graph depict the entities, and the edges show the relationship between the entities. The figure below shows an example:

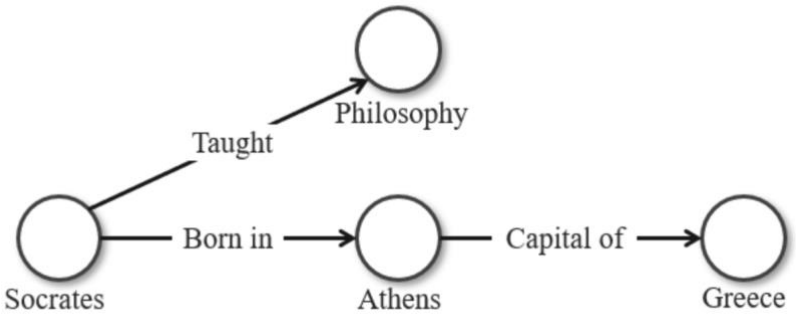


Fig 1. Example knowledge graph

A. Rules for relation extraction

There can be different rules for extracting entities and relations from text. Depending on these rules, different types of relations can be extracted. In the current project, subject-predicate-object triplets have been extracted from sentences with exactly one subject and one object. The spacy library has been used for parsing the dependency tree (ref_dependency_tree) of each sentence. The focus is on sentences with a verb as the root of the dependency tree. Example:

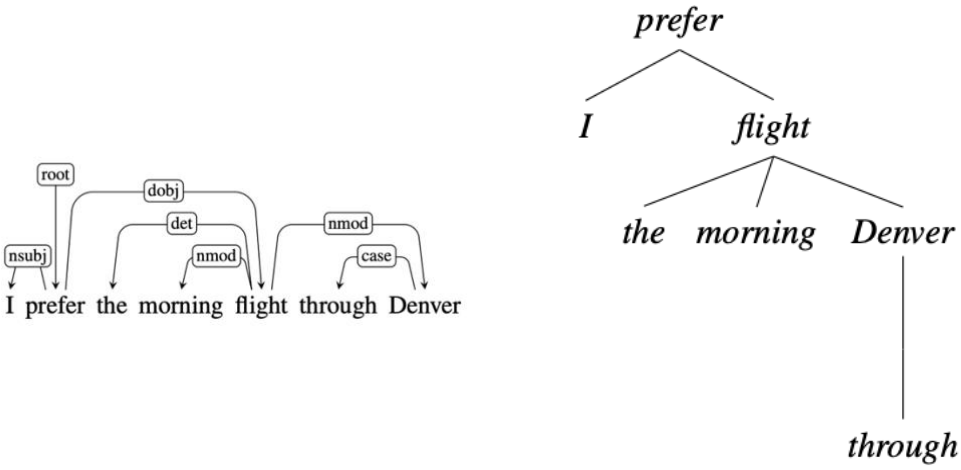


Fig 2. Example dependency tree

In the above example, "prefer" is the verb which is also the root of the sentence. The subject "I" (this will be resolved using coreference resolution) is on the left side of the tree, and the object "flight" is on the right. Relations from sentences of this type that have a subject on the left and an object on the right will be extracted. Defining other rules to extract relations of the type: object-predicate-subject may also be done at a later stage of the project.

B. Capturing better nodes and relations

The nodes in the knowledge graph may span multiple words. For example, in fig. 2 above, we want to capture "morning flight" and not just "flight" as one of the nodes. The relation could then be, "I (node 1) -> prefer (edge) -> morning flight (node 2)", which is better than capturing the relation, "I (node 1) -> prefer (edge) -> flight (node 2)". This is handled by retokenising and merging noun chunks and named entities as a single token before extracting subjects and objects from the sentence. Noun chunks are "base noun phrases", flat phrases with a noun as their head. You can think of noun chunks as a noun plus the words describing the noun. For example, "the lavish green grass" or "the world's largest tech fund". Spacy has its own entity extracting pipeline, which will be used.

To capture better relations from prepositional phrases like "She **walked to** his desk**",** we will extract the *adposition* ("to") which comes after the verb. We will also capture phrasal verbs by extracting *particle* after the verb. Ex. "He turned **on** the light", in this example, "on" is the particle. Extraction of verbs from prepositional phrases and phrasal verbs will be done using part-of-speech tagging and dependency tree.

C. Algorithm

i) Pre-process the text and resolve coreference

```
    # preprocess text
    text = re.sub(r'\n+', '.', text)  # replace multiple newlines with period
    text = re.sub(r'\[\d+\]', ' ', text)  # remove reference numbers
    text = nlp(text)
    if coref:
        text = nlp(text._.coref_resolved)  # resolve coreference clusters
```

Coreference resolution: Coreference resolution is finding the set of mentions in the text that refer to the same thing. Reference link: https://web.stanford.edu/~jurafsky/slp3/21.pdf. Example: Elon Reeve Musk is an entrepreneur and business magnate. He is the founder, CEO and Chief Engineer at SpaceX.

In the above two sentences, "Elon Reeve Musk" and "He" refer to the same person, and having this information beforehand would help us create better knowledge graphs. In the example sentence, without coreference resolution, we would not know the founder of SpaceX.

Neuralcoref, a pipeline extension for Spacy which resolves conference clusters using a neural network, is used in the project. Along with word embeddings, the authors create custom features to give context information and train the model on the OntoNotes_corpus dataset. For the current task, this pre-trained model is used to resolve coreferences in the news articles.

Brief description of Neuralcoref model from the official blog (huggingface_neuralcoref):

"Our model then goes very roughly as follow: we take word embeddings for several words inside and around each mention, average them if needed, add some simple integer features (length of the mention, speaker information, location of the mentions…) to obtain a features representation for each mention and its surroundings. Then we plug these representations into two neural nets. A first neural net gives us a score for each pair of a mention and a possible antecedent, while a second neural net gives us a score for a mention having no antecedent (sometimes a mention is the first reference to an entity in a text). We can then simply compare all these scores together and take the highest score to determine whether a mention has an antecedent and which one it should be."

The figure below shows this model:

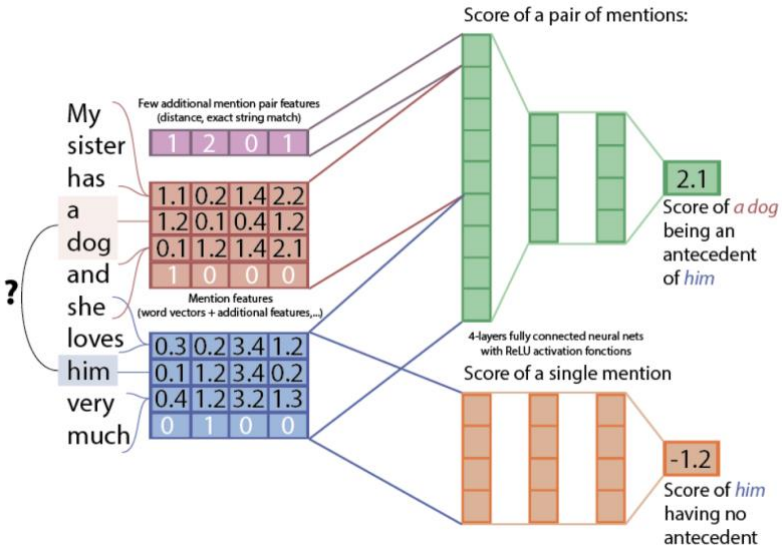


Fig 3. Neuralcoref model

ii) Split text into sentences and merge entities and noun chunks spanning across multiple tokens into a single token

```
45  sentences = [sent.string.strip() for sent in text.sents]  # split text into sentences
46  ent_pairs = []
47  for sent in sentences:
48      sent = nlp(sent)
49      spans = list(sent.ents) + list(sent.noun_chunks)  # collect nodes
50      spans = spacy.util.filter_spans(spans)
51      with sent.retokenize() as retokenizer:
52          [retokenizer.merge(span, attrs={'tag': span.root.tag,
53                                          'dep': span.root.dep}) for span in spans]
```

Line 49 collects all entities and noun chunks in a sentence and stores them in a list. Line 50 then filters out the spans that overlap. In case of an overlap, the span with the longest length is kept, and the shorter one is removed. Finally, lines 51 to 53 assign a single token to the spans that we want to process as a single token. This new token is assigned the same part-of-speech and dependency tag as the token (in the span) with the shortest path to the root of the sentence. This allows us to preserve tags like subject and object over something like a modifier.

iii) If the sentence has more than 1 subject or object, we will move to the next sentence.

```
55
56      deps = [token.dep_ for token in sent]
57
58      # limit our example to simple sentences with one subject and object
59      if (deps.count('obj') + deps.count('dobj')) != 1\
60              or (deps.count('subj') + deps.count('nsubj')) != 1:
61          continue
```

iv) Identify the subject and object in the sentence.

```
63      for token in sent:
64          if token.dep_ not in ('obj', 'dobj'):  # identify object nodes
65              continue
66          subject = [w for w in token.head.lefts if w.dep_
67                     in ('subj', 'nsubj')]  # identify subject nodes
```

We go through each token in the sentence and check if it is an object. To extract the subject placed on the object's left side in the dependency tree (we are extracting sentences of the type subject->predicate-> object), we move to the object's head and check all its children for the subject. In figure 2 b, we would identify "flight" as the object. To get the subject, we would then move to the head of "flight", which is "prefer", and search for all its children on the left for the subject. This will give us "I" as the subject, which will be resolved due to coreference resolution in step i.

v) After locating the subject, extract the root verb of the sentence from the dependency tree. This root token will be the relation between object and subject.

```
68              if subject:
69                  subject = subject[0]
70                  # identify relationship by root dependency
71                  relation = [w for w in token.ancestors if w.dep_ == 'ROOT']
```

vi) Add adposition or particle to the relation (motivation in section 1.B).

```
72                  if relation:
73                      relation = relation[0]
74                      # add adposition or particle to relationship
75                      if relation.nbor(1).pos_ in ('ADP', 'PART'):
76                          relation = ' '.join((str(relation), str(relation.nbor(1))))
77                  else:
78                      relation = 'unknown'
```

Adposition or particle in the sentence will be the next neighbour of the root verb. We check if the next neighbour is any of these and combine it with the root verb. This helps capture better relations from prepositional phrases and phrasal verbs.

vii) Refine the subject and object captured

The noun chunks which are being treated as single tokens may have some unwanted part-of-speech words. We will check the subject and object captured for such unwanted tokens.

```
18  def refine_ent(ent, sent):
19      unwanted_tokens = (
20          'PRON',  # pronouns
21          'PART',  # particle
22          'DET',  # determiner
23          'SCONJ',  # subordinating conjunction
24          'PUNCT',  # punctuation
25          'SYM',  # symbol
26          'X',  # other
27      )
28      ent_type = ent.ent_type_  # get entity type
29      if ent_type == '':
30          ent_type = 'NOUN_CHUNK'
31          ent = ' '.join(str(t.text) for t in nlp(str(ent)) if t.pos_ not in unwanted_tokens and t.is_stop == False)
```

Example:
Noun chunk before refining: “These greenhouse gases”
Noun chunk after refining: “greenhouse gases”

If the extracted subject or object is a CARDINAL or ORDINAL, we will extract the words following it in the sentence till we encounter a verb or punctuation.

Example:
extracted subject before refining: “one”
extracted subject after refining: “one of the editors”

Finally, all the extracted triplets are stored in a dataframe. The dataframe is saved as "kg_coref.csv".

2. Sentiment Analysis

Sentiment analysis is the task of predicting how positive or negative a text may sound to a reader. The input here will be the news article, and the output will be a score from -1 (highly negative reaction) to +1 (highly positive reaction). I used the FLAIR library to get sentiment for the BBC news articles. FLAIR is an NLP framework designed to facilitate training and distribution of state-of-the-art sequence labelling, text classification and language models. (FLAIR: https://github.com/flairNLP/flair)

FLAIR uses a transformer-based model for predicting sentiment. Transformer models have encoder-decoder stacks with attention mechanism (reference: Attention is all you need). FLAIR's sentiment classifier is trained on movie and product reviews. Unlike rule-based models for sentiment analysis like VADER, which only cares about individual words, transformer models capture the context in which words appear. Transformer based models show two major benefits over RNNs:

i) Theoretically, the attention mechanism allows the transformer model to have an infinite reference window compared with smaller windows in RNNs.
ii) Transformer based models take less training time.

Algorithm

i) Split text into sentences using FLAIR's sentence splitter

```
150         # initialize sentence splitter
151         splitter = SegtokSentenceSplitter()
152
153         # use splitter to split text into list of sentences
154         try:
155             sentences = splitter.split(text)
156         except:
157             print("sentence could not be split")
158             print(text)
```

ii) Load flair's text sentiment predictor (transformer based) and predict sentiment for sentences.

```
160           # load tagger call predict
161           classifier = TextClassifier.load('sentiment')
162           classifier.predict(sentences)
```

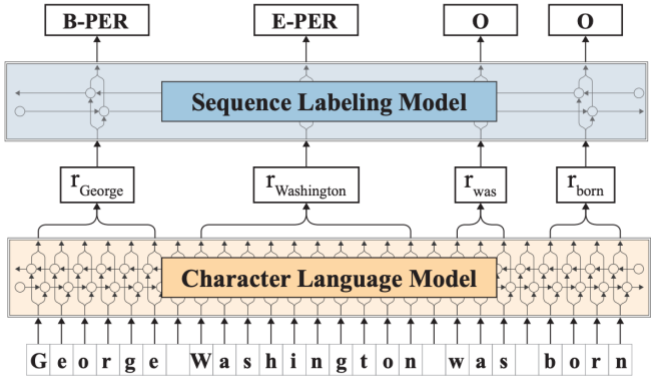
iii) Iterate through the sentences in article and calculate the average sentiment.

```
170         # iterate through sentences and print predicted labels
171         sentiment = 0
172         for s in sentences:
173             sentiment =  sentiment + s.labels[0].score * (-1,1)[str(s.labels[0]).split()[0].startswith("POS")]
174
188
189         sentiment = sentiment/len(sentences)
```

Line 173 adds the predicted sentiment for all sentences. Line 189 normalises the sum of sentiments with the total number of sentences in the article.

3. Named Entity Recognition

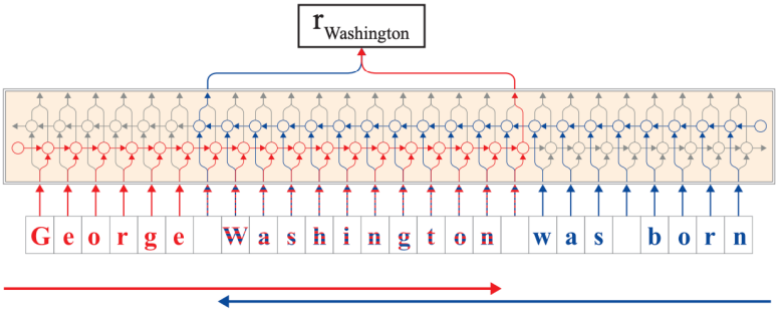
Named entity recognition is done using the FLAIR framework (ref: NER_paper). A graphical representation of the model is shown below:



Each sentence is passed as a sequence of characters to a bidirectional character-level neural language model, from which internal character states are retrieved for each word to create a contextual string embedding. This embedding is then utilized in the BiLSTM-CRF sequence tagging module for Named Entity recognition.

The embeddings are taken from hidden states of both forward and backward LSTM models while learning to predict the next character based on previous characters. The generated word-level embeddings are shown to be highly effective in downstream sequence labelling tasks.

To get the word embedding, the hidden state after the last character in the word is extracted from the forward model. Similarly, the hidden state before the word's first character is extracted from the backward model. This is shown in the figure below:



The final word embeddings are passed into a BiLSTM-CRF sequence labelling module for Named entity recognition. The BiLSTM-CRF model consists of a Bidirectional LSTM layer which takes in the word embeddings. On top of this BiLSTM layer, we have a Conditional Random Field layer that predicts the NER tags. This model is trained on the CONLL03 dataset.

Algorithm:

i) Split sentences, load pre-trained NER model and predict

```
150         # initialize sentence splitter
151         splitter = SegtokSentenceSplitter()
152 
153         # use splitter to split text into list of sentences
154         try:
155             sentences = splitter.split(text)
156         except:
157             print("sentence could not be split")
158             print(text)
159 
160         # predict tags for sentences
161         tagger = SequenceTagger.load('ner')
162         tagger.predict(sentences)
```

ii) Go over the sentences and collect NER tags for the article.

```
167 
168         ner_tag_dict = {}
169 
170         # iterate through sentences and print predicted labels
171         for s in sentences:
172 
173             s_dict = s.to_dict(tag_type='ner')
174 
175             for idx in range(len(s_dict['entities'])):
176                 entity_text = s_dict['entities'][idx]['text']
177                 for idx2 in range(len(s_dict['entities'][idx]['labels'])):
178                     entity_dict = s_dict['entities'][idx]['labels'][idx2].to_dict()
179                     entity_name = entity_dict['value']
180                     entity_conf = entity_dict['confidence']
181                     if(entity_conf>0.8):
182                         if entity_name not in ner_tag_dict:
183                             ner_tag_dict[entity_name] = [entity_text]
184                         else:
185                             ner_tag_dict[entity_name].append(entity_text)
```

In the above code lines, 181 to 185 check the confidence of the prediction. If this confidence is above 80%, the NER category is stored. Otherwise, we ignore it.

4. Web scraping

Python and Selenium are used to scrap all articles listed on https://www.bbc.co.uk/news/science-environment-56837908. A dataframe is created to store the information. Each row of the dataframe will correspond to a single article. The dataframe will have the following columns: url of the page, author of article, date of publishing, heading, text in the article, sentiment of the text, Named entities in the article, caption of images, url of images and location of images stored on local disk.

A class WebMine() will be created which will have dataframe as a member variable and various methods to extract the fields listed above. This will help in easy extension to other websites that the user might want to scrap in the future.

```
15 class WebMine():
16 
17     def __init__(self, df):
18         # initialize with already existing dataframe
19         self.df = df
```

Algorithm:

i) Initialize an instance of WebMine class.

```python
if __name__ == "__main__":

    # try loading df, if unsuccesful create empty df
    try:
        df = pd.read_csv("bbc_df.csv")
    except:
        cols = ['page_url','author','date','heading','text','text_sentiment','NER','img_caption','img_url','img_loc']
        df = pd.DataFrame(columns=cols)

    bbc = WebMine(df)
```

ii) Collect all links to articles from https://www.bbc.co.uk/news/science-environment-56837908.

```python
    def get_links(self,url,xpath):

        driver = webdriver.Chrome()
        driver.get(url)
        my_Xpath = xpath
        all_elements = driver.find_elements(By.XPATH, my_Xpath)

        # Fetch and store the links
        links = []
        for element in all_elements:
            links.append(element.get_attribute('href'))

        return links
```

Selenium provides various methods to locate elements in a page. To get links we can navigate the page structure using xpath. After locating the element that contains link for articles, the links can be extracted from this element by its href attribute.

iii) Extract information from dataframes

```python
    def get_page_df(self,links):

        driver = webdriver.Chrome()
        # Loop through all the links and launch one by one

        itr = 0
        for link in links:
            print("scraping: ",link)

            # check if link has already been scraped before
            if link in self.df.values:
                print("link already exists")
                continue

            # store attributes in a dict and append to df in the end
            dict_link = {}

            # open link
            driver.get(link)
            dict_link['page_url'] = link

            # get author name
            dict_link = self.get_author_name(driver,dict_link)
            # get heading
            dict_link = self.get_heading(driver,dict_link)
            # get text
            dict_link = self.get_text(driver,dict_link)
            # get image
            dict_link = self.get_image(driver,dict_link,link,itr)
            # get time
            dict_link = self.get_time(driver,dict_link)

            # get sentiment and NER
            if(dict_link['text']!=''):
                sentiment, ner_tag_dict = self.get_NER_Sent(dict_link['text'])
                dict_link['text_sentiment'] = sentiment
                dict_link['NER' ] = ner_tag_dict

            time.sleep(5)
            self.df = self.df.append(dict_link, ignore_index=True)

            itr = itr+1

        return
```

We can loop over the links extracted in step ii and get all the required information using the get_page_df() function. The given BBC link has articles that do not change often. Inside the loop, we first check if the current link has already been scraped before.

All the functions used within this get_page_df() navigate the page structure using xpath or id (functions: get_author_name, get_heading, get_text, get_image, get_time). Various checks have been placed within these functions to ensure correct information is being extracted. The function get_NER_sent() calculates the sentiment from the article's text and returns named entities. This is done using the algorithm defined in sections 2 and 3.

There are many articles that do not have an author name mentioned. For example: https://www.bbc.co.uk/news/science-environment-59649066. In such cases, the author column is left empty. Article https://www.bbc.co.uk/news/resources/idt-d6338d9f-8789-4bc2-b6d7-3691c0e7d138 has a different structure compared to all other articles. This article has been omitted for now.

iv) Save dataframe

The final step is to remove any duplicates from the dataframe and save it. The dataframe is saved as "bbc_df.csv".

5. Web application

Flask, Python's web development framework, will be used to create a web application to visualise the sentiment of BBC articles collected over time. Along with Flask, Bokeh will be used to create and embed figures into the served HTML in real-time. One of the great features of Bokeh is the ability to export a figure as raw HTML and JavaScript. This allows us to inject figures that are created programmatically into a Flask application's templates.

Algorithm:

i) Create a Bokeh object

```
45      source = ColumnDataSource()
```

The object source created above is used to represent data in a way that is standardized for Bokeh elements. We will pass our data into that object, which is fed to the Bokeh figure. It is structured as an object, mapping keys to an array of values. This object can be directly accessed and manipulated using CustomJS.

ii) Load the scraped dataframe containing information about BBC articles.

```
27      # get dataframe and define bokeh object
28      def selectedMovies():
29          df = pd.read_csv('bbc_df.csv').reset_index()
30          df = df[df['text_sentiment'].notna()]
31          df = df.drop_duplicates(subset='page_url')
32
33          df['list_NER'] = ''
34
35          for index, row in df.iterrows():
36              list_NERs = []
37              row_ner = eval(row['NER'])
38              for key in list(row_ner.keys()):
39                  list_NERs = list_NERs + list(np.unique(row_ner[key]))
40              df.at[index,'list_NER'] = list(np.unique(list_NERs))
41
42          res = df.to_dict('records')
43          return res
44
45      currArticles = selectedMovies()
```

selectedMovies() function loads the dataframe, filters out rows that do not have text sentiment value and removes duplicates. Lines 33 to 40 collect all unique NERs from the dataframe. This list of NERs will be used as a control for the interactive Bokeh figure.

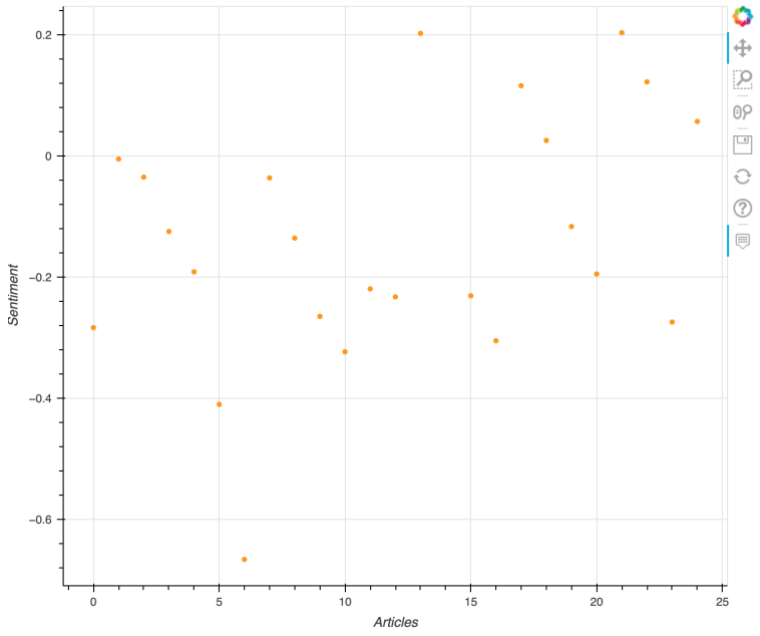
iii) Fill values in the source object created in step 1. Data is stored as key value pairs taken from the dataframe.

```
49      source.data = dict(
50              x = [d['index'] for d in currArticles],
51              y = [d['text_sentiment'] for d in currArticles],
52              color = ["#FF9900" for d in currArticles],
53              heading = [d['heading'] for d in currArticles],
54              date = [pd.to_datetime(d['date'], yearfirst = True) for d in currArticles],
55              NER = [d['list_NER'] for d in currArticles],
56              url = [d['page_url'] for d in currArticles]
57          )
```

iv) Next, we will make a bokeh figure to display the data in source object

```
106     fig = figure(plot_height=600, plot_width=720 , tooltips=[ ("Heading", "@heading"),("url", "@url") ,("NER", "@NER")])
107
108     fig.circle(x="x", y="y", source=source, size=5, color="color", line_color=None)
109     fig.xaxis.axis_label = "Articles"
110     fig.yaxis.axis_label = "Sentiment"
```

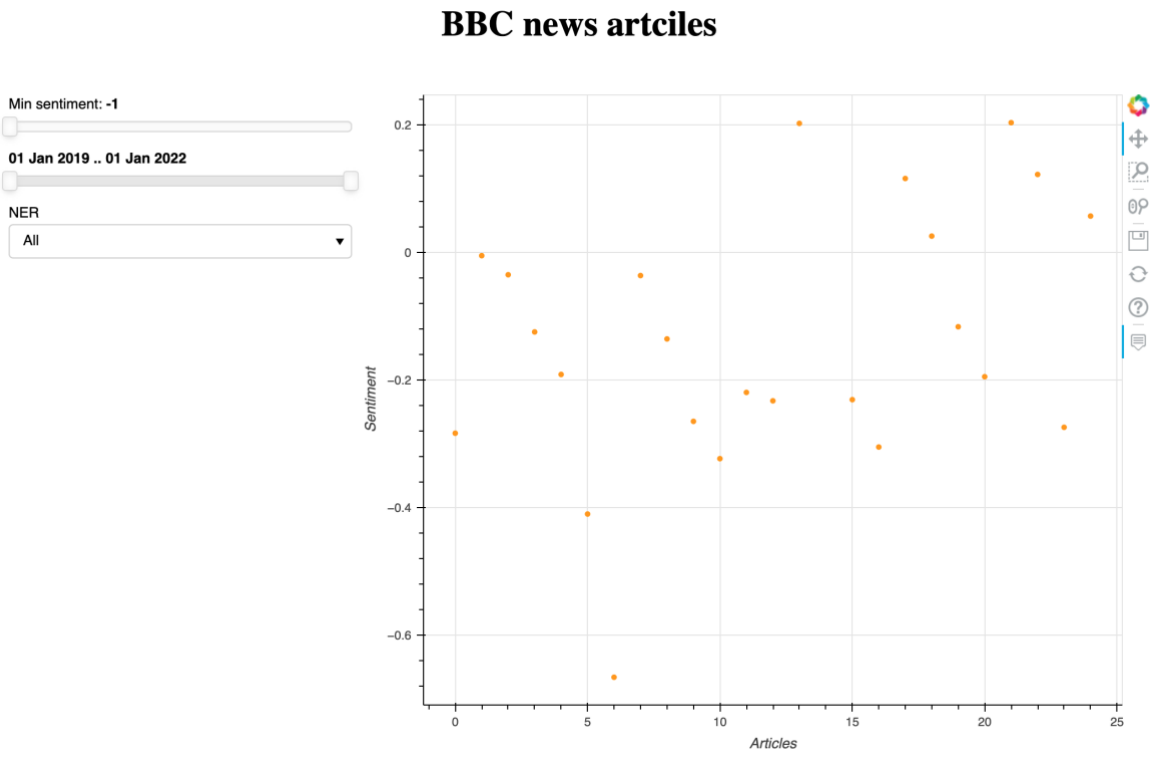
The above code will make a Bokeh figure with x-axis as the index of articles from the dataframe, and the y-axis will display the article sentiment. Tooltips defined in line 106 allows us to set information to display when we hover over a particular article. Note that all these fields have been defined in the source object. This will generate the below plot:



v) Add UI widgets

```
59      # define controls
60      NER_list = []
61      for idx in range(len(currArticles)):
62          NER_list = NER_list + currArticles[idx]['list_NER']
63      NER_list = list(np.unique(NER_list + ['All']))
64
65      controls = {
66          "min_sentiment": Slider(title="Min sentiment", value=-1, start=-1, end=1, step=0.1),
67
68          "date_range_slider": DateRangeSlider(value=(date(2019, 1, 1), date(2022, 1, 1)),
69                                               start=date(2019, 1, 1), end=date(2022, 1, 1)),
70
71          "NER": Select(title="NER", value="All", options=NER_list)
72      }
```

Next, we will add some UI widgets that users can interact with to manipulate the data. Three controls are defined in the code above. Line 66 defines a slider control based on the sentiment value. Based on the slider, articles above the threshold will be filtered and displayed. Line 68-69 define a widget to filter articles based on date, and line 71 will filter based on NER values. The dropdown menu for NER filtering has all unique NERs from dataframe. The UI after adding widgets looks like this:



vi) Define JS callback

Next, we need to implement a callback that occurs when any of our controls change. This is accomplished with Bokeh's CustomJS module.

```
76      callback = CustomJS(args=dict(source=source, controls=controls), code="""
77          if (!window.full_data_save) {
78              window.full_data_save = JSON.parse(JSON.stringify(source.data));
79          }
80      |
81          var full_data = window.full_data_save;
82          var full_data_length = full_data.x.length;
83          var new_data = { x: [], y: [], color: [], heading: [], date: [], NER: [], url: [] }
84
85          for (var i = 0; i < full_data_length; i++) {
86              if (full_data.y[i] === null || full_data.date[i] === null || full_data.NER[i] === null)
87                  continue;
88              console.log((controls.date_range_slider.value[1] > full_data.date[i]) && (controls.date_range_slider.value[0] <
    full_data.date[i]))
89
90              if (
91                  full_data.y[i] > controls.min_sentiment.value &&
92                  (controls.NER.value === 'All' || full_data.NER[i].includes(controls.NER.value, 0)) &&
93                  controls.date_range_slider.value[0] < full_data.date[i] &&
94                  controls.date_range_slider.value[1] > full_data.date[i]
95              ) {
96                  Object.keys(new_data).forEach(key => new_data[key].push(full_data[key][i]));
97              }
98          }
99
100
101
102         source.data = new_data;
103         source.change.emit();
104     """)
```

The above callback function will be called every time any of the inputs are changed. Lines 77-78 will save a global JavaScript variable when this function is called for the first time. This variable will not change henceforth. Lines 85 to 98 go through each row in the data frame and check if the condition defined by UI widgets is satisfied. All articles that satisfy the widget conditions will be returned to be displayed on the web app.

vii) Create an HTML file for flask application

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>Document</title>
7       {{ js_resources|indent(4)|safe }}
8       {{ css_resources|indent(4)|safe }}
9       {{ plot_script|indent(4)|safe }}
10  </head>
11  <body>
12      <h1 style="text-align: center; margin-bottom: 40px;">BBC news artciles</h1>
13      <div style="display: flex; justify-content: center;">
14          {{ plot_div|indent(4)|safe }}
15      </div>
16  </body>
17  </html>
```

This is a simple HTML webpage with four attributes: js_resources, css_resources, plot_script, and plot_div. Bokeh is going to give us all the variables that we will pass into these four attributes.

viii) Define layout and inject elements into HTML.

```
115     inputs_column = column(*controls_array, width=320, height=1000)
116     layout_row = row([ inputs_column, fig ])
117
118     script, div = components(layout_row)
119     return render_template(
120         'index.html',
121         plot_script=script,
122         plot_div=div,
123         js_resources=INLINE.render_js(),
124         css_resources=INLINE.render_css(),
125     )
```

Line 115-116 creates a column of our controls called input_controls, followed by a row of input_controls and our figure. Here's a mapping of what is being injected into our HTML template.

- plot_script: Figure-specific JavaScript
- plot_div: Figure-specific HTML wrapped in tag
- js_resources: General Bokeh-required JavaScript
- css_resources: General Bokeh-required CSS

To run the web application:

- Go to BBC_articles folder
- run : "export FLASK_APP=flask_app.py && flask run"