DSCI-511 SP-2022 Final Project

The New York Times Archive API for Text Analysis Dataset

Alec Peterson
ap3842@drexel.edu

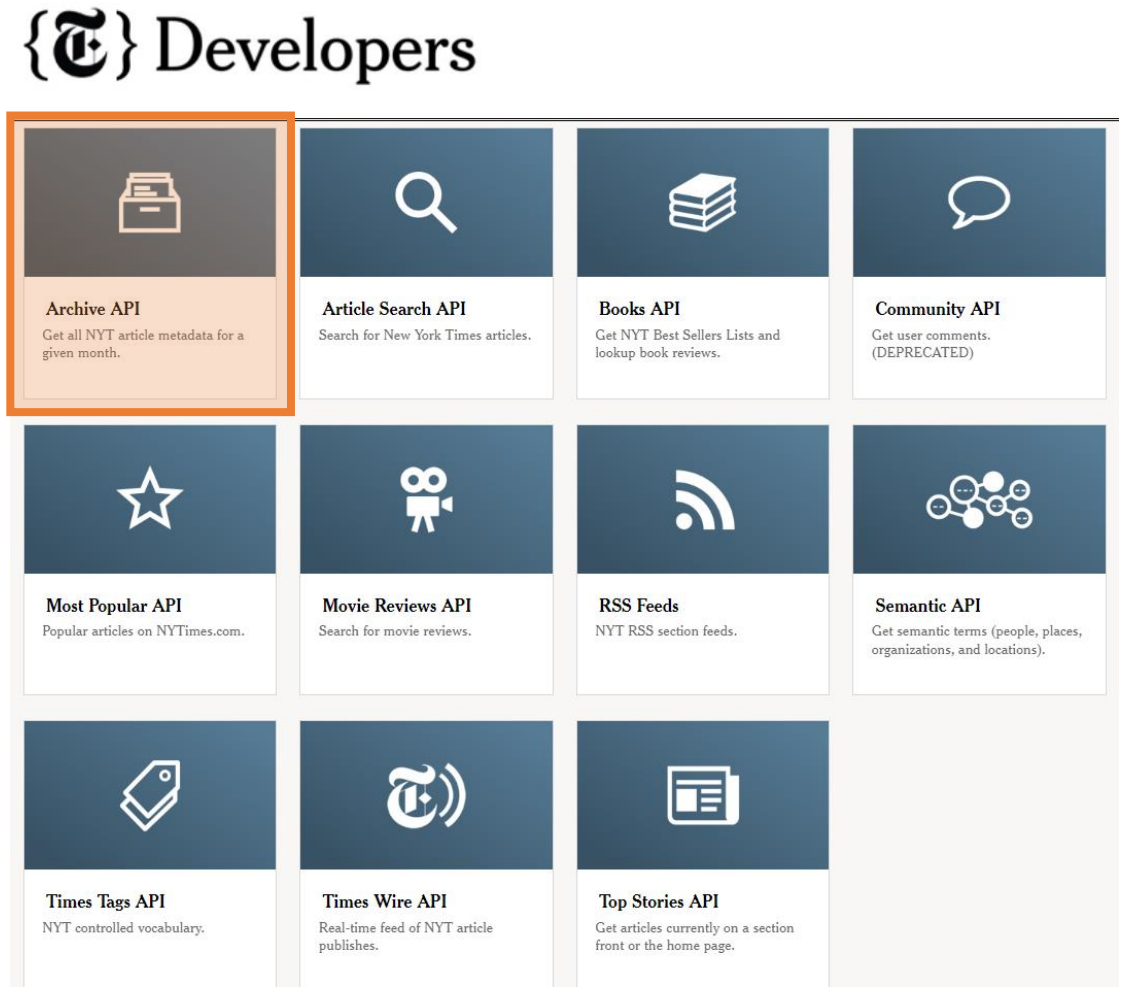
23-May-2022

Motivations

The New York Times

- NYT has retained articles dating back to the organization's founding 1851
- Articles information accessible in JSON form via REST-ful Archive API
- Gives metadata for a given (year, month)

- Use cases:
  - **Natural Language Processing** (via headlines, abstracts) - topic modeling, sentiment analysis…
  - **Historical analyses** – key events
  - **Quantitative analyses** – wordcount, article count, word and entity frequency, temporal comparisons…
  - **Meta analyses of *The New York Times* organization** – prolific contributors, output by section…)
  - **Article database**



API Response - Structure

The New York Times

```
# Define a function to generate a url given input year number (year_no) and month number (month_no, a number 1 through 12)
def gen_url(year_no, month_no):

    url = "https://api.nytimes.com/svc/archive/v1/{}/{}.json?api-key={api_key}".format(str(year_no),
                                                                                        str(month_no),
                                                                                        api_key = "(your_key_here)")


    return url
```

```
# Get a response object using the generated url:
response = requests.get(gen_url(2000, 5))
results = response.json()

results["response"]["docs"][0]
```

API Response - Structure

The New York Times

```python
# Define a function to generate a url given input year number (year_no) and month number (month_no, a number 1 through 12)
def gen_url(year_no, month_no):

    url = "https://api.nytimes.com/svc/archive/v1/{}/{}.json?api-key={api_key}".format(str(year_no),
                                                                                        str(month_no),
                                                                                        api_key = "(your_key_here)")


    return url
```

```python
# Get a response object using the generated url:
response = requests.get(gen_url(2000, 5))
results = response.json()

results["response"]["docs"][0]
```

- results *(dict)*
  - copyright *(str)*
  - response *(dict)*
    - docs *(list)*
    - meta *(dict)*
      - Hits *(int)*

API Response - Structure

The New York Times

```python
# Define a function to generate a url given input year number (year_no) and month number (month_no, a number 1 through 12)
def gen_url(year_no, month_no):

    url = "https://api.nytimes.com/svc/archive/v1/{}/{}.json?api-key={api_key}".format(str(year_no),
                                                                                        str(month_no),
                                                                                        api_key = "(your_key_here)")


    return url
```

```python
# Get a response object using the generated url:
response = requests.get(gen_url(2000, 5))
results = response.json()

results["response"]["docs"][0]
```

- results *(dict)*
  - copyright *(str)*
  - response *(dict)*
    - docs *(list)*
    - meta *(dict)*
      - Hits *(int)*

Each element in docs represents the metadata for an article

API Response - Structure

The New York Times

- results *(dict)*
  - copyright *(str)*
  - response *(dict)*
    - docs *(list)*
    - meta *(dict)*
      - Hits *(int)*

- **abstract** (str): Abstract of the article
- **web_url** (str): URL
- **snippet** (str): Short description often repeat of **abstract**
- **lead_paragraph** (str): First paragraph in the article
- **print_section** (str): Letter denoting print section
- **print_page** (str): String representation of print page number
- **source** (str): Organization providing article source
- **multimedia** (list → dict…): List of dictionaries for associated media types (e.g. images)
- **headline** (dict): Article headline
  - **main** (str): Headline text
  - **kicker** (str): (often None)
  - **content_kicker** (str): (often None)
  - **print_headline** (str): Printed headline, often same as Article headline
  - **name**: (often None)
  - **seo**: (often None)
  - **sub**: (often None)
  - **keywords** (list → dict…): dictionaries with keywords
- **pub_date** (str): string representation of publication date (formatted like datetime)
- **document_type** (str): usually “article”
- **news_desk** (str): e.g. “Foreign”, “Investigative”, “Express”, “Learning”…
- **section_name** (str): e.g. “U.S.”, “World”, “Science”…
- **byline** (dict): contributor information
  - **original** (str): “e.g. “By Jane Smith” or organization
  - **person** (list → dict…): list of dictionaries with contributor first, last, etc
  - **organization**: contributing organization
- **type_of_material** (str): Description of article type e.g. “News”, “Review”
- **_id** (str): unique ID in NYT database
- **word_count** (int): word count of the article
- **uri** (str): often same as _id

API Response - Structure

The New York Times

- results *(dict)*
  - copyright *(str)*
  - response *(dict)*
    - docs *(list)*
    - meta *(dict)*
      - Hits *(int)*

- **abstract** (str): Abstract of the article
- **web_url** (str): URL
- **snippet** (str): Short description often repeat of **abstract**
- **lead_paragraph** (str): First paragraph in the article
- **print_section** (str): Letter denoting print section
- **print_page** (str): String representation of print page number
- **source** (str): Organization providing article source
- **multimedia** (list → dict…): List of dictionaries for associated media types (e.g. images)
- **headline** (dict): Article headline
  - **main** (str): Headline text
  - **kicker** (str): (often None)
  - **content_kicker** (str): (often None)
  - **print_headline** (str): Printed headli
  - **name**: (often None)
  - **seo**: (often None)
  - **sub**: (often None)
  - **keywords** (list → dict…): dictionaries with keywords
- **pub_date** (str): string representation of publication date (formatted like datetime)
- **document_type** (str): usually "article"
- **news_desk** (str): e.g. "Foreign", "Investigative", "Express", "Learning"…
- **section_name** (str): e.g. "U.S.", "World", "Science"…
- **byline** (dict): contributor information
  - **original** (str): "e.g. "By Jane Smith" or organization
  - **person** (list → dict…): list of dictionaries with contributor first, last, etc
  - **organization**: contributing organization
- **type_of_material** (str): Description of article type e.g. "News", "Review"…
- **_id** (str): unique ID in NYT database
- **word_count** (int): word count of the article
- **uri** (str): often same as _id

Useful for NLP

Allows for temporal analyses, indexing and .groupby() day /month /year

Useful for quantitative analyses e.g. .groupby() and .count()

Unique index for each article – allows for joins of disparate tables

Generating Initial Dataframe

`gen_url(year_no, month_no)`

Generates the URL for an API call for a given integer year and month combination

```
# Define a function to generate a url given input year number (year_no) and month number (month_no, a number 1 through 12)
def gen_url(year_no, month_no):

    url = "https://api.nytimes.com/svc/archive/v1/{}/{}.json?api-key={api_key}".format(str(year_no),
                                                                                        str(month_no),
                                                                                        api_key = "(your_key_here)")

    return url
```

Generating Initial Dataframe

`gen_url(year_no, month_no)` Generates the URL for an API call for a given integer year and month combination

`doc_df(docs)`

1. **Make a dictionary with a key for each variable of interest and an empty list as a value**
2. **Iterate over the list of article dictionaries in `result[response][docs]`, appending each article's info to its corresponding list in the dictionary from 1.**
3. **Return a `pd.DataFrame` of the dictionary**

```python
def doc_df(docs):
    import pandas as pd
    from collections import defaultdict

    doc_keys = ["pub_date", "abstract", "web_url",
                "lead_paragraph","document_type", "news_desk",
                "section_name", "type_of_material", "source",
                "headline", "byline", "_id",
                "word_count"]

    doc_dict = defaultdict(list)
```

```python
    for key in doc_keys:
        for doc in docs:

            # headline is a dict
            if key == "headline":
                doc_dict[key].append(doc[key]["main"])

            # get "original" string instead of dealing with "person" dict
            elif key == "byline":
                doc_dict[key].append(doc[key]["original"])

            # all other keys
            else:
                doc_dict[key].append(doc[key])

    return pd.DataFrame(doc_dict)
```

Generating Initial Dataframe

`gen_url(year_no, month_no)`

Generates the URL for an API call for a given integer year and month combination

`doc_df(docs)`

1. Make a dictionary with a key for each variable of interest and an empty list as a value
2. Iterate over the list of article dictionaries in `result[response][docs]`, appending each article's info to its corresponding list in the dictionary from 1.
3. Return a `pd.DataFrame` of the dictionary

`parse_docs(year_no, month_no)`

Calls `gen_url()` to generate a URL, makes a request to NYT Archive API with that URL, then calls `doc_df()` to generate a dataframe

```
def parse_docs(year_no, month_no):

    # Generate the url
    url = gen_url(year_no, month_no)
    response = requests.get(url)
    results = response.json()
    docs = results["response"]["docs"] # a list of documents, each item is a dictionary

    # Generate the dataframe
    df = doc_df(docs)

    return df
```

Generating Initial Dataframe

`gen_url(year_no, month_no)`

Generates the URL for an API call for a given integer year and month combination

`doc_df(docs)`

1. Make a dictionary with a key for each variable of interest and an empty list as a value
2. Iterate over the list of article dictionaries in `result[response][docs]`, appending each article's info to its corresponding list in the dictionary from 1.
3. Return a `pd.DataFrame` of the dictionary

`parse_docs(year_no, month_no)`

Calls `gen_url()` to generate a URL, makes a request to NYT Archive API with that URL, then calls `doc_df()` to generate a dataframe

`make_df_NYTarchive(year_start, year_end)`

Generates a big dataframe with info from all articles between `year_start` and `year_end`

Generating Initial Dataframe

`make_df_NYTarchive(year_start, year_end)`

Generates a big dataframe with info from all articles between `year_start` and `year_end`

1. **Takes a start year and end year (exclusive) then iterates over the list of years and all months (1 – 12), calls `parse_docs()` on each combination and appends the resulting df to a list (`df_list`)**
   - **Has to wait 6 seconds between each call as API calls cannot exceed 10 calls/minute**

```python
def make_df_NYTarchive(year_start, year_end):

    # Timer start
    start_time = timeit.default_timer()

    # Collect dataframes for each year and month, calling the parse_docs function
    df_list = []
    for year in range(year_start, year_end):
        for month in range(1, 13):
            # Start timer
            parse_time_start = timeit.default_timer()

            # Call parse_docs, then append the resulting dataframe to df_list
            df = parse_docs(year, month)
            df_list.append(df)

            # End timer
            parse_time_end = timeit.default_timer()

            # Wait at least 6 seconds between API calls
            parse_time_total = parse_time_end - parse_time_start
            if parse_time_total > 6:
                continue
            else:
                time.sleep(6 - parse_time_total)
```

Generating Initial Dataframe

`make_df_NYTarchive(year_start, year_end)`

Generates a big dataframe with info from all articles between `year_start` and `year_end`

1. Takes a start year and end year (exclusive) then iterates over the list of years and all months (1 – 12), calls `parse_docs()` on each combination and appends the resulting df to a list (`df_list`)
   - Has to wait at least 6 seconds between each call as API calls cannot exceed 10 calls/minute
2. **Combines dataframes in `df_list`, reindexes, makes "`pub_date`" a date, then adds columns for "`year`" and "`month`"**

```
# Make a "big" dataframe combining the dataframes for each year-month's articles
df_big = df_list[0]
for i in range(1, len(df_list)):
    df_big = pd.concat([df_big, df_list[i]])

## Reindex the big dataframe
df_big.index = pd.RangeIndex(len(df_big))

## Change pub_date to date, add "year" and "month" column
df_big["pub_date"] = pd.to_datetime(df_big["pub_date"])
df_big["pub_date"] = df_big["pub_date"].apply(lambda x: x.date())
df_big["year"] = df_big["pub_date"].apply(lambda x: x.year)
df_big["month"] = df_big["pub_date"].apply(lambda x: x.month)

## Reorder the columns
df_big = df_big.loc[:, ["_id", "pub_date", "year", "month", "headline", "abstract", "lead_paragraph", "byline",
                        "word_count", "document_type", "news_desk", "section_name", "type_of_material", "source", "web_url"]]
```

Generating Initial Dataframe

`make_df_NYTarchive(year_start, year_end)`

Generates a big dataframe with info from all articles between `year_start` and `year_end`

1. Takes a start year and end year (exclusive) then iterates over the list of years and all months (1 – 12), calls `parse_docs()` on each combination and appends the resulting df to a list (`df_list`)
   - Has to wait at least 6 seconds between each call as API calls cannot exceed 10 calls/minute
2. Combines dataframes in `df_list`, reindexes, makes "`pub_date`" a date, then adds columns for "`year`" and "`month`"
3. **Returns resulting dataframe with reordered columns, and prints out elapsed time to execute the function (in minutes)**

```
    # Timer elapsed
    elapsed = timeit.default_timer() - start_time
    print("Elapsed time: {} minutes".format(elapsed/60))
    return df_big
```

Generating Initial Dataframe

`make_df_NYTarchive(year_start, year_end)`

Generates a big dataframe with info from all articles between `year_start` and `year_end`

1. Takes a start year and end year (exclusive) then iterates over the list of years and all months (1 – 12), calls `parse_docs()` on each combination and appends the resulting df to a list (`df_list`)
   - Has to wait at least 6 seconds between each call as API calls cannot exceed 10 calls/minute
2. Combines dataframes in `df_list`, reindexes, makes "`pub_date`" a date, then adds columns for "`year`" and "`month`"
3. **Returns resulting dataframe with reordered columns, and prints out elapsed time to execute the function (in minutes)**

```
# Timer elapsed
elapsed = timeit.default_timer() - start_time
print("Elapsed time: {} minutes".format(elapsed/60))
return df_big
```

$$\boldsymbol{N} = (year\_end) - (year\_start)$$

$$\frac{\sim 2\ min}{year\ of\ articles} * (\boldsymbol{N}\ years) = \sim(2\ to\ 3)\boldsymbol{N}\ minutes\ per\ query$$

A decade's worth of articles could take ~20 minutes to load, owing much to the API calls/minute limitations

Storage as .parquet

- Apache Parquet files (.parquet) are a columnar storage format that have more efficient storage and can be read in faster than the equivalent .csv file
- Parquet files can be efficiently read and written in Python with the `pyarrow` package, which takes advantage of a language-independent in-memory columnar data Apache Arrow format
  - `pandas` can also read in Parquet files using `pandas.read_parquet()`, which is faster than `pandas.read_csv()` (by ~3x)

```
def save_as_parquet(year_start, year_end):
    import pyarrow as pa
    import pyarrow.parquet as pq

    df_big = make_df_NYTarchive(year_start, year_end)

    table = pa.Table.from_pandas(df_big)
    pq.write_table(table ,"{}_to_{}.parquet".format(year_start, year_end))

    print("Dataframe saved as {}_to_{}.parquet".format(year_start, year_end))
    return df_big
```

Storage as .parquet

- Apache Parquet files (.parquet) are a columnar storage format that have more efficient storage and can be read in faster than the equivalent .csv file
- Parquet files can be efficiently read and written in Python with the `pyarrow` package, which takes advantage of a language-independent in-memory columnar data Apache Arrow format
  - `pandas` can also read in Parquet files using `pandas.read_parquet()`, which is faster than `pandas.read_csv()` (by ~3x)
- In addition to `pyarrow` and `pandas`, the `ibis` and `polars` packages in Python and the {`arrow`} package in R can take advantage of the .parquet and Arrow in-memory formats for fast analytics on large tabular datasets

spaCy for Text Pre-Processing

- The spaCy package offers an efficient way to perform text pre-processing, including (but not limited to):
  - **Tokenization**: breaking text up into smaller chunks
  - **Lemmatization**: converting a word into its "base" form, with linguistic context (e.g."meeting" (*verb*) becomes "meet", while "meeting" (*noun*) stays "meeting")
  - **Named Entity Recognition**: Recognizing people, places, organizations, figures, etc.
  - **Filtering out**: stopwords and punctuation
- Structured representations that take advantage of spaCy features might include:
  - Adding "tokens", "lemmas", and/or "entities" list columns for columns that contain relevant text (`headline`, `abstract`, `lead_paragraph`)
  - Unnesting (`using df.explode()`) those to make a row for each token / lemma / entity

Analyses of this "tabular-ized" data are more approachable via `pandas` (for data manipulation) and `matplotlib` / `seaborn` (for visualization)

spaCy for Text Pre-Processing

- The spaCy pipeline by default includes:
  - part-of-speech tagger
  - dependency parser
  - named entity recognizer
- Speed of `nlp.pipe()` to turn text into a spaCy doc object will increase if non-essential components are disabled

```
[1]: import spacy
     nlp = spacy.load("en_core_web_sm")

[2]: nlp.pipeline

[2]: [('tagger', <spacy.pipeline.pipes.Tagger at 0x164a2ee4b50>),
      ('parser', <spacy.pipeline.pipes.DependencyParser at 0x164a2eba8e0>),
      ('ner', <spacy.pipeline.pipes.EntityRecognizer at 0x164a2eba9a0>)]

[3]: import spacy
     nlp = spacy.load("en_core_web_sm", disable=["tagger", "parser"])

[4]: nlp.pipeline

[4]: [('ner', <spacy.pipeline.pipes.EntityRecognizer at 0x164a346d040>)]
```

spaCy Operations – make into doc objects

1. Make a list of spaCy doc objects for text columns `headline`, `abstract`, and `lead_paragraph` columns

```
# Make list of spaCy doc objects from text columns
headline_docs = list(nlp.pipe(df_big["headline"]))

abstract_docs = list(nlp.pipe(df_big["abstract"]))

lead_para_docs = list(nlp.pipe(df_big["lead_paragraph"]))
```

Using the `n_processes` argument can take advantage of multithreading (parallelization) to improve speed

(~40-50% faster for 2 processes on my computer, but diminishing returns after that)

spaCy Operations- make dictionary with lemmas and entities

2. Iterate over the list and add to a dictionary, with article `_id` as the key, and the values a dictionary with tokens and entities

Remove stop words, punctuation, spaces. Subjectively, removing anything that resembles numbers (e.g. “10” and “ten” etc.)

```
article_dict = {}
for i in range(len(df_big["_id"])):
    article_dict[df_big["_id"][i]] = {"pub_date": df_big["pub_date"][i],

                                      # headline and its tokens, and entities
                                      "headline": df_big["headline"][i],

                                      "lemmas_headline": [token.lemma_ for token in headline_docs[i]
                                                          if not token.is_stop and not token.is_punct and not token.is_space and not token.like_num],
                                      "entities_headline": list(headline_docs[i].ents),

                                      # abstract and its tokens and entities
                                      "abstract": df_big["abstract"][i],

                                      "lemmas_abstract": [token.lemma_ for token in abstract_docs[i]
                                                          if not token.is_stop and not token.is_punct and not token.is_space and not token.like_num],
                                      "entities_abstract": list(abstract_docs[i].ents),

                                      # lead paragraph and its tokens and entities
                                      "lead_paragraph": df_big["lead_paragraph"],

                                      "lemmas_lead_para": [token.lemma_ for token in lead_para_docs[i]
                                                           if not token.is_stop and not token.is_punct and not token.is_space and not token.like_num],
                                      "entities_lead_para": list(lead_para_docs[i].ents)}
```

spaCy Operations – make a nested dataframe

3. Turn `article_dict` into a dataframe, use `.transpose()` to make the article `_id` key into the index

```
df2_big = pd.DataFrame(article_dict).transpose()
df2_big["pub_date"] = pd.to_datetime(df2_big["pub_date"])
df2_big["pub_date"] = df2_big["pub_date"].apply(lambda x: x.date())

df2_big.head()
```

	pub_date	headline	lemmas_headline	entities_headline	abstract	lemmas_abstract	entities_abstract	lead_paragraph	lemmas_lead_para	entities_lead_para
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	[Bridge]	[]	LEAD: One of the many sad bridge stories of 19...	[LEAD, sad, bridge, story, concern, final, Aus...	((One), (1989), (the, Australian, National, Te...	0 One of the many sad bridge stories of ...	[sad, bridge, story, concern, final, Australia...	[(One), (1989), (the, Australian, National, Te...
nyt://article/0a0e2668-b979-56d1-b675-c7f521131236	1990-01-01	He Has Tyson On His Mind	[Tyson, Mind]	[]	LEAD: THOSE on Donovan Ruddock's Christmas-car...	[LEAD, Donovan, Ruddock, Christmas, card, list...	((Donovan, Ruddock, 's), (Christmas), (this, y...	0 One of the many sad bridge stories of ...	[Donovan, Ruddock, Christmas, card, list, get,...	[(Donovan, Ruddock, 's), (Christmas), (this, y...
nyt://article/0a71f8ee-d649-5ae8-9524-e2f1d6d5a0aa	1990-01-01	For Dinkins, Pomp, Ceremony, Triumph And a Dre...	[Dinkins, Pomp, Ceremony, Triumph, Dream, Real...	[(Dinkins), (Pomp)]	LEAD: The walls of David N. Dinkins's borough ...	[LEAD, wall, David, N., Dinkins, borough, pres...	((David, N., Dinkins, 's), (dozens), (City, Ha...	0 One of the many sad bridge stories of ...	[wall, David, N., Dinkins, borough, president,...	[(David, N., Dinkins, 's), (dozens), (City, Ha...
nyt://article/0da7cd16-c122-51c0-b392-2484ac4b7de3	1990-01-01	Army Doesn't Have to Compete With Marines; W...	[Army, Compete, Marines, Need, Navy]	[(Army), (Navy)]	LEAD: To the Editor:	[LEAD, Editor]	()	0 One of the many sad bridge stories of ...	[Editor]	[]

spaCy Operations – Define function to unnest

4. Define `make_exploded_df(df, col_name)` that takes a dataframe and a desired list column name and makes each element a row via `df.explode()`

```
def make_exploded_df(df, col_name):
    if col_name == "lemmas_headline" or "entities_headline":
        base_col = "headline"
    elif col_name == "lemmas_abstract" or "entities_abstract":
        base_col = "abstract"
    else:
        base_col = "lead_paragraph"

    df_exp = df.explode(column = col_name).loc[:, ["pub_date", base_col, col_name]]

    return df_exp
```

spaCy Operations – unnest via lemmas (`headline`)

4. Define `make_exploded_df(df, col_name)` that takes a dataframe and a desired list column and makes each element a row via `df.explode()`

Test out on headline lemmas:

```
df_lems_headline = make_exploded_df(df2_big, "lemmas_headline")
df_lems_headline.head()
```

	pub_date	headline	lemmas_headline
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	Bridge
nyt://article/0a0e2668-b979-56d1-b675-c7f521131236	1990-01-01	He Has Tyson On His Mind	Tyson
nyt://article/0a0e2668-b979-56d1-b675-c7f521131236	1990-01-01	He Has Tyson On His Mind	Mind
nyt://article/0a71f8ee-d649-5ae8-9524-e2f1d6d5a0aa	1990-01-01	For Dinkins, Pomp, Ceremony, Triumph And a Dre...	Dinkins
nyt://article/0a71f8ee-d649-5ae8-9524-e2f1d6d5a0aa	1990-01-01	For Dinkins, Pomp, Ceremony, Triumph And a Dre...	Pomp

spaCy Operations – unnest via lemmas (`lead paragraph`)

4. Define `make_exploded_df(df, col_name)` that takes a dataframe and a desired list column name and makes each element a row via `df.explode()`

Test out on lead paragraph lemmas:

```
df_lems_lead_para = make_exploded_df(df2_big, "lemmas_lead_para")
df_lems_lead_para.head()
```

	pub_date	headline	lemmas_lead_para
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	sad
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	bridge
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	story
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	concern
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	final

spaCy Operations – unnest via entities (`abstract`)

4. Define `make_exploded_df(df, col_name)` that takes a dataframe and a desired list column name and makes each element a row via `df.explode()`

Test out on abstract entities:

```
df_ents_abs = make_exploded_df(df2_big, "entities_abstract")
df_ents_abs.head()
```

	pub_date	headline	entities_abstract
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	(One)
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	(1989)
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	(the, Australian, National, Team, Championship)
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	(Canberra)
nyt://article/065d970c-0342-5066-a441-f59423263e3d	1990-01-01	Bridge	(February)

```
df_ents_abs["entities_abstract"][2]
```

```
the Australian National Team Championship
```

While stored as a tuple, accessing the elements gives the named entities

Conclusions

- The lemma-unnested or entity-unnested dataframes can be analyzed themselves, or joined (via the `_id` column) to the larger dataframe if desired
- Combinations of these dataframes could be used to identify key people or figures throughout history, key events, or key words to identify topics
- Challenges:
  - Loading speed due to API call limitations
  - spaCy object information lost when storing in `pandas` dataframes (coerced to “object” data type)
- Future Directions:
  - Implement rule-based strategies for stopword removal, entity recognition
  - Additional multithreading for spaCy and Python operations
  - Use `pyarrow` string representations in `pandas` dataframe
  - Analyze longer text objects i.e. actual articles or book reviews using other NYT APIs and prepare datasets for topic modeling or sentiment analysis