NOUN PHRASE VISUALIZATION: THE COMPLETE PROCESS

PART 1: INDEXING OF NOUN PHRASE FILES (90278 FILES)

Created Python Module: index_noun_phrases.py

Prerequisites:

- An existing Solr index of the arXiv metadata of 90278 research papers. This index is used to get the published date for each of the research papers. This index has been built from the arXiV XML file and its documentation is available as part of the documentation for the PaperSearch search engine. The published date is a date range field, i.e. its field type is as below.

```
<fieldType name="daterange" class="solr.DateRangeField"/>
<field name="published_date" type="daterange" indexed="true"
stored="true" multiValued="false"/>
```

- 90278 noun phrase files (from the 90278 research papers), each of which has gone through a preprocessing/cleaning step by an army of sed commands. These sed commands have already described in the SedCommandList.txt document.

Implementation details and code explanation:

The first step is to index each noun phrase in each of the 90278 noun phrase files. However, indexing every individual noun phrase in a file (with duplicates) will result in a very large index and will also make it difficult for later retrieval. So, this program uses a Counter from the collections module to count duplicate noun phrases in a file. For 1 noun phrase file, there are N entries inserted into the index (where N is the no. of unique noun phrases in a file).

The code snippet for the insert_into_solr, which uses Pysolr, is shown below. Here, after creating a connection to the empty Solr collection 'nounphrases', the iglob function of the glob module is used to iterate through the *.nps.txt files in the given directory. As mentioned in the PaperSearch Solr document, Pysolr expects a list of dictionaries to be passed in its 'add' function.

For each noun phrase file that is read, the filename is obtained and the arxiv identifier is extracted from it. This is used to query the arxiv_metadata Solr index and get the published date (this is explained later).

Now, the noun phrase files are of the format: *noun phrase* \t *start offset* \t *end offset.*

We read each line and keep only the noun phrase as we are not interested in the offsets. A new collections.Counter object is created at this stage, which counts the number of occurrences of each phrase in a single file. These frequencies are inserted into the solr_content along with the arxiv identifier (from the filename), the noun phrase itself and the published date. This dictionary is inserted into a list to be sent to solr.

As this list will become very large if it stores the dictionaries for all noun phrases in all files, it is sent to Solr for each file and reinitialized with an empty list.

```
def insert_into_solr():
    """ Inserts records into an empty Solr index which has already been
created. It inserts frequencies of each noun phrase per file along with the
arxiv identifier (from the file name) and the published date (obtained from
the arxiv_metadata Solr index). """
```

```
    solr = pysolr.Solr('http://localhost:8983/solr/nounphrases')
    folderpath = '/home/ashwath/Files/NPFiles'

    for filepath in iglob(os.path.join(folderpath, '*.nps.txt')):
        # Insert all the phrases in a file into Solr in a list
        list_for_solr = []
        with open(filepath, "r") as file:
            # Get the filename without extension (only 1st 2 parts
            # of filename after splitting)
            filename= os.path.basename(filepath)
            filename = '.'.join(filename.split('.')[0:2])
            # published date is a default dict with lists as values.
            published_date = search_solr(filename, 'arxiv_metadata',
'arxiv_identifier')
            # Line is tab-separated (phrase, start, end). We want only
phrase Don't add useless phrases to list 'phrases'. Use a generator
expression instead of a list comprehension
            phrases = (line.split("\t")[0].lower().strip() for line in file
                if line.split("\t")[0].lower().strip() != "")
            temp_phrase_counter = Counter(phrases)
            for phrase, frequency in temp_phrase_counter.items():
                solr_content = {}
                solr_content['phrase'] = phrase
                solr_content['num_occurrences'] = frequency
                solr_content['published_date'] = published_date
                solr_content['arxiv_identifier'] = filename
                list_for_solr.append(solr_content)

        # Upload to Solr file by file
        solr.add(list_for_solr)
```

Getting the published date from the arxiv_metadata Solr collection is quite straightforward, as shown in the following code snippet. It performs an exact match query on the 'arxiv_identifier' search field, and then parses the json to get only the first published date. There may be multiple published dates in the arxiv XML file, where the later dates correspond to revisions. However, for our visualization, we are only interested in the original published date. Note also that only one row will be returned for an arxiv identifier in this index.

```
def search_solr(query, collection, search_field):
    """ Searches the arxiv_metadata collection on arxiv_identifier (query)
    and returns the published date which it obtains from
parse_arxiv_json"""
    solr_url = 'http://localhost:8983/solr/' + collection + '/select'
    # Exact search only
    query = '"' + query + '"'
    url_params = {'q': query, 'rows': 1, 'df': search_field}
    solr_response = requests.get(solr_url, params=url_params)
    if solr_response.ok:
        data = solr_response.json()
        return parse_arxiv_json(data)
    else:
        print("Invalid response returned from Solr")
        sys.exit(11)

def parse_arxiv_json(data):
    """ Parses the response from the arxiv_metadata collection in Solr
    for the exact search on arxiv_identifier. It returns the published date
of the first version of the paper (i.e. if there are multiple versions, it
    ignores the revision dates)"""
```

```
    docs = data['response']['docs']
    # There is only one result returned (num_rows=1 and that is the nature
of the data, there is only one paper with 1 arxiv identifier). As a JSON
array is returned and we only want the first date, we take only the first
element of the array.
    return docs[0].get('published_date')[0]
```

PART 2: CREATING A JSON FILE WITH NO. OF PHRASES & DOCUMENTS PER MONTH.

Created Python Modules: pickle_nounphrase_data.py, groupcount_noun_phrases.py

Prerequisites: Same as Part 1, Parts 1 and 2 can be run in parallel.

Implementation details and code explanation:

The purpose of these 2 programs (modules) is to calculate the number of phrases and number of documents for each month, based on the published date from the arxiv_metadata Solr index. This is required because the graphs in Part 3 find the frequency of a noun phrase in a month compared to all other noun phrases (i.e. what percentage of the total noun phrases is occupied by a particular noun phrase) and the no. of documents in which a noun phrase per month compared to all the other noun phrases (again, calculated a percentage of the no. of documents in the month)

The **pickle_nounphrase_data.py** program iterates through the 90278 files, creates a data frame with the total no. of noun phrases in each file -- the data frame contains the filename (arxiv identifier), the published date (obtained as in Part 1 from the arxiv_metadata index) and the number of phrases in the file. It pickles this data frame so that it can be used by the groupcount_noun_phrases.py module to get the monthly phrase and document frequencies.

```
def count_phrases():
    """Creates a data frame from all the noun phrase files. The data frame
contains the filename, no. of phrases in each file and the published date
from the arxiv xml file. This dataframe is finally pickled."""
    df = pd.DataFrame(columns=['filename', 'published_date',
'num_phrases'])
    basepath = '/home/ashwath/Files/NPFiles'
    # Initialize to -1 as we want to insert from loc[0] into the dataframe.
    file_num = -1
    for filepath in iglob(os.path.join(basepath, '*.nps.txt')):
        file_num += 1
        with open(filepath, 'r') as file:
            # Get the filename without extension (only 1st 2 parts of
filename after splitting)
            filename= os.path.basename(filepath)
            filename = '.'.join(filename.split('.')[0:2])
            published_date = search_solr(filename, 'arxiv_metadata',
'arxiv_identifier')
            # Get the line count: this will give the total no. of noun
phrases (there is one noun phrase in each line)
            # All the lines are normalized, empty lines have already been
removed using sed in pre-processing.
            for line_num, line in enumerate(file):
                pass
        # print(file_num)
        df.loc[file_num] = {'filename':filename,
'published_date':published_date, 'num_phrases': line_num + 1}
    # Pickle the dataframe
```

```
    pickle_temp = open("total_phrase_counter.pickle", "wb")
    pickle.dump(df, pickle_temp)
    pickle_temp.close()
```

The **groupcount_noun_phrases.py** module takes a pickle file containing a Pandas data frame of file wise total phrase frequencies. Each row in this data frame is identified by a unique arxiv_identifier and contains a published date field. The data in this data frame is grouped by month and 2 aggregates: sum (corresponding to the no. of phrases in that month) and count (corresponding to the no. of documents in that month) are calculated. The dates are converted into the format required in Part 3, and these are then converted to a JSON array of objects: one object for each of the above 2 aggregates. This array of objects is finally stored in a JSON file. A code snippet is shown with very detailed inline comments.

```
def create_monthly_grouped_df():
    """Unpickle the dataframe which contains filename, published date and
num_phrases, group by published date, and calculate a count and a sum on
num_phrases. Return the aggregates in a data frame"""
    # Unpickle data frame
    pickle_in = open('total_phrase_counter.pickle', "rb")
    df = pickle.load(pickle_in)
    # Change the index of the dataframe from the autogenerated index to the
column published_date
    df.set_index('published_date', inplace=True)
    # Change the index to a datetime index
    df.index = pd.DatetimeIndex(df.index)
    # Create a new dataframe which is formed by grouping on the
published_date index by month and calculating 2 aggregates: 1. sum (which
gives the no. of noun phrases in a given month) and 2. count (which gives
the no. of documents in  a given month).  This also resets the index as a
column and suitably renames columns.
    grouped_df =
df.groupby(pd.Grouper(freq='1M')).num_phrases.agg(['sum','count']).rename(c
olumns={'sum': 'monthly_phrasefreq',
'count': 'monthly_docfreq'})
    # We don't need the day, only the month and the year -- format year-
month. E.g. 2018-08.
    grouped_df.index = grouped_df.index.strftime('%Y-%m')
    # We need to transpose the df before calling to_json because it takes
the column names as the keys of the JSON (same as to_dict). After taking
the transpose, the index values (date with year and month) become the
columns and hence the keys in the JSON. Write this to a JSON file: the json
file will have a json array (python list) with 2 objects (dicts): one for
the total no. of phrases in each month, and one for the total no. of
documents in each month.
    grouped_df.T.to_json('phrases_and_docs_monthly.json', orient='records')
```

PART 3: VISUALIZATION OF NOUN PHRASES

The following description talks about MONTHLY noun phrase visualization. Yearly noun phrase visualization is described in a short addendum at the end.

Created Python Modules: nounphrase_visualization_monthly.py

Prerequisites:

- The Solr nounphrases index created in Part 1 should already have been created.
- The Json files with the monthly document and phrase frequencies (created in Part 2) should be present in the same directory.

This module uses the Solr index and json file which have been created in Parts 1 and 2, and builds a user interface along with 2 graphs. For this, it uses Plotly's Dash Framework.

Quick Dash Overview:

Dash is a Python framework which is built on top of Plotly.js, React.js, and Flask. It allows the programmer to focus on building analytics/visualizations and takes care of the Javascript using React. External CSS may be provided, and Dash's html components are used to write html within the program.

All the graphs are plotly graphs, and the plotly syntax can be used within the Dash code.

The code to create the graphs has to be inserted in a Python decorator function, which triggers a callback whenever an event takes place (for example, when a 'Submit' button is clicked, or when a keypress is released). This callback can have only one output component (a Html div element), but multiple inputs are normally allowed. If a Submit button is used, a 'State' component is also passed to the callback with the html id of the respective input element.

Implementation steps and code explanation:

Step 1:

Imports are needed for:

(a) Getting the data from Solr: requests
(b) Aggregation, storing intermediate data: pandas
(c) User interface: dash.dependencies, dash_html_components as html
(d) Graphs: plotly.graph_objs, dash_core_components as dcc

Step 2:

2 separate graphs are to be plotted for one input element for this use case. Before defining the decorator functions and the logic to get data based on the query from Solr, some Html elements need to be created for the inputs and outputs. Other things like labels are also created at this stage, and external style sheets are supplied. Dash's recommended external style sheet as well as another style sheet to hide an undo button are included in this case.

```
app = dash.Dash()

# Add the default Dash CSS, and some custom (very simple) CSS to remove the
undo button
app.css.append_css({'external_url':
'https://codepen.io/chriddyp/pen/bWLwgP.css'})
```

```
app.css.append_css({'external_url':
'https://rawgit.com/lwileczek/Dash/master/undo_redo5.css'})
colours = {
    'background': '#111111',
    'text': '#0080A5'
}
app.layout = html.Div(style={'backgroundColor': colours['background'],
                             'height':'100vh', 'width': '100%'},
                      children=[
    html.H2(children='Distribution of Noun phrases over time',
            style={
                'textAlign': 'center',
                'color': colours['text']
            }
    ),
    html.Label('Graph these comma-separated noun phrases: ',
               style={
                'textAlign': 'left',
                'color': colours['text'],
                'fontSize': '1.4em'
            }),
    dcc.Input(id='npinput1-state', value='', type='text'),
    html.Button(id='submit-button', n_clicks=0, children='Submit'),
    html.Div(id='output_total'),
    html.Div(id='output_unique')

])
```

After initializing an object of the Dash class, all the HTML components are defined inside an app.layout element, which is a top-level html Div element.

In the above code, we have an input element created using dcc as well as a submit button, an html label and two output divs (one for each graph). Inline CSS is included where necessary. CSS ids need to be defined for the input and output elements: the text box, the submit button and the output divs so that they can be modified/accessed by the decorator function.

Step 3:

Variables or functions which need to be used in the graphs need to be in the global scope. In our case, we need to take the user's queries (separated by noun phrases) and for each query, we will have to get frequency data from Solr, load it into a Pandas data frame and create intermediate data frames by carrying out various operations, and final data frames.

The code snippet below queries the 'nounphrases' collection and returns a list of dictionaries.

```
def search_solr_parse_json(query, collection, search_field):
    """ Searches the nounphrases collection on 'phrase' (query),
    parses the json result and returns it as a list of dictionaries where
    each dictionary corresponds to a record.
    ARGUMENTS: query, string: the user's query entered in a search box
               (if it is comma-separated, only one part of the query is
sent
               to this function).
               collection: the Solr collection name (=nounphrases)
               search_field: the Solr field which is queried (=phrase)
    RETURNS: docs, list of dicts: the documents (records) returned by Solr
             AFTER getting the JSON response and parsing it."""
    solr_url = 'http://localhost:8983/solr/' + collection + '/select'
```

```
    # Exact search only
    query = '"' + query + '"'
    # for rows, pass an arbitrarily large number.
    url_params = {'q': query, 'rows': 100000, 'df': search_field}
    solr_response = requests.get(solr_url, params=url_params)
    if solr_response.ok:
        data = solr_response.json()
        docs = data['response']['docs']
        return docs
    else:
        print("Invalid response returned from Solr")
        sys.exit(11)
```

The following snippet creates a Pandas data frame from the list of dictionaries, removes unnecessary columns and sets published_date as the index.

```
def dataframe_from_solr_results(documents_list):
    """ Takes a list of dictionaries (each dictionary is a record) obtained
by parsing the JSON results from Solr, converts it into a dataframe, and
keeps only the 4 important columns (discards _version_ and id, and also
phrase, keeps published_date, num_occurrences and arxiv_identifier).
Finally, it makes sure that the published_date is the new index.
    ARGUMENTS: documents_list, list of dicts: list of documents (records)
returned by Solr for one search query
    RETURNS: docs_df, Pandas dataframe, the Solr results converted into a
Pandas dataframe with index=published_date, columns=arxiv_identifier,
num_occurrences"""
    docs_df = pd.DataFrame(documents_list)
    # Remove phrase too, as all the rows will have the same value
    # (the solr query field was phrase).
    docs_df.drop(['_version_', 'id', 'phrase'], axis=1, inplace=True)
    # Change the published_date column from Solr's string timestamp format
to a pandas
    # datetime object with just dates.
    docs_df.published_date = pd.to_datetime(docs_df.published_date)
    # Make sure the published_date is the index. Once it is the index, we
don't
    # really need the column any more.
    docs_df.set_index('published_date', inplace=True, drop=True)
    return docs_df
```

The following function is used to group the data frame by month and create 2 new data frames: 1 for the total no. of occurrences (monthly), and another for the total no. of documents in which a phrase appears (monthly). As shown below, any NaN values obtained (there will be NaNs for months in which the user's chosen phrase doesn't appear) are changed to 0, and the datatype is set to int for the num_occurrences column.

```
def calculate_aggregates(docs_df):
    """ Takes a Pandas data frame with index=published_date, cols:
num_occurrences and arxiv_identifier as input, calculates the no. of unique
and total occurrences by grouping by the month and year part of
published_date and then calculating the count and sum based on the column
num_occurrences. The aggregate results are suitably renamed and the
published_date index is reset so that it becomes a column in the output
dataframe. 2 dataframes (unique counts and total counts) are returned.
    ARGUMENTS: docs_df, Pandas dataframe with index=published_date,
               columns=num_occurrences and arxiv_identifier
```

```
    RETURNS: docs_df_total, a Pandas df grouped on published_date month and
year, on which 'sum' is applied on num_occurrences.
             docs_df_unique, a Pandas df grouped on published_date month
and year, on which 'count' is applied on num_occurrences.
             IMPORTANT: the returned dfs have sum and count in the same
column called num_occurrences, a new sum/count column is not created.
"""
    # Drop arxiv_identifier, we want to group by the published_date index,
and aggregate on num_occurrrences.
    docs_df.drop('arxiv_identifier', axis=1, inplace=True)
    # Dataframe 1 takes the sum of num_occurrences after grouping by month
(and year)
    docs_df_total = docs_df.groupby(pd.Grouper(freq='1M')).sum()
    # docs_df_total.index has day as well, we keep only month and year
    # Change num_occurrences to int after replacing nan by 0
    docs_df_total.num_occurrences =
docs_df_total.num_occurrences.fillna(0).astype('int64')
    # Dataframe 2 takes the count of num_occurrences after grouping by
month (and year). This is a monthly documnet frequency
    docs_df_unique = docs_df.groupby(pd.Grouper(freq='1M')).count()
    # Change num_occurrences to int after replacing nan by 0
    docs_df_unique.num_occurrences =
docs_df_unique.num_occurrences.fillna(0).astype('int64')
    return docs_df_total, docs_df_unique
```

The following function is used to normalize the num_occurrences in each of the data frames by the total no. of phrases in each month (this applies to the docs_df_total data frame) and the total no. of documents in each month (this applies to the docs_df_unique data frame).

The json file has to be read and each of the 2 records (it contains an array of 2 records) is assigned to a separate dictionary. A new column is then created in each of the data frames as a Period object (datatype) – this column contains the year and month in exactly the same format as the json file (e.g. 2018-08).

The pandas map function is used to get the values corresponding to the month in each row of the data frame from the respective dictionary. The num_occurrences is divided by the value from the dict, and multiplied by 100 to convert it to a percentage.

```
def get_percentage_aggregates(docs_df_total, docs_df_unique):
    """ This function takes 2 dataframes -- one has monthly phrase
frequencies, the other has monthly document frequencies -- and normalizes
the values by dividing by total no. of phrases in the corresponding months
and total no. of documents in the corresponding month respectively (and
multiplies by 100) to get percentages.
    ARGUMENTS: docs_df_total, a Pandas df grouped on published_date month
and year, on which 'sum' is applied on num_occurrences.
             docs_df_unique, a Pandas df grouped on published_date month
and year, on which 'count' is applied on num_occurrences.
    RETURNS: docs_df_total, the data frame in the arguments with an
additional field 'percentage_occurrences' calculated by dividing the
current value for each month by the no. of phrases in that month
             docs_df_unique, the data frame in the arguments with an
additional field 'percentage_occurrences' calculated by dividing the
current value for each month by the no. of docs in that month
    NOTE: The total no. of docs/phrases in each month is present in a json
file phrases_and_docs_monthly.json """
```

```
    # Read the Json file which has the monthly total phrases and documents
-- 2 Json objects in a
    # json array. Assign each object to a dictionary.
    with open('phrases_and_docs_monthly.json', 'r') as file:
        json_array= json.load(file)
    # json_array is a list of 2 dicts.
    monthly_phrases_total = json_array[0]
    monthly_docs_total = json_array[1]
    # For each of the dataframes, create a monthyear column in the format
year-month, e.g. 2017-08.
    # This is a string and matches the value from the json file.
    # Create monthyear column as a period object with frequency = every
month
    docs_df_total['monthyear'] = docs_df_total.index.to_period('M')
    # Convert the period object to a string
    docs_df_total.monthyear =  docs_df_total.monthyear.astype('str')
    # Create a new column which uses the value in the monthyear string
column as a key in the monthly_phrases_total dict,and gets the
corresponding value. The no. of occurrencesis divided by this number. The
na_action is not strictly necessary, it is just a precaution which inserts
NaN if a key (month+year) is not found. Finally, NaNs are produced if the
dict value has a 0 (divide by 0). These NaNs are replaced by 0. * 100
because the final result is in %.
    docs_df_total['percentage_occurrences'] = (100 *
docs_df_total.num_occurrences / docs_df_total['monthyear']
        .map(monthly_phrases_total, na_action=None)).fillna(0)
    # Repeat the process for docs_df_unique
    docs_df_unique['monthyear'] = docs_df_unique.index.to_period('M')
    # Convert the period object to a string
    docs_df_unique.monthyear =  docs_df_unique.monthyear.astype('str')
    docs_df_unique['percentage_occurrences'] = (100 *
docs_df_unique.num_occurrences / docs_df_unique['monthyear']
        .map(monthly_docs_total, na_action=None)).fillna(0)
    return docs_df_total, docs_df_unique
```

Finally, all these operations are wrapped in an outer function, which can be called by the Dash decorator functions. It is important to note that the get_aggregated_data function returns the appropriate data frames to the calling Dash functions if data is found for a Solr index, but it returns None whenever a query fetches no data from the Solr index.

```
def get_aggregated_data(query):
    """ Function which returns an aggregated function for a valid query and
    None for an invalid one.
    ARGUMENTS: query, string, one of the parts of the user's comma-
separated query
    RETURNS: docs_df_total, a Pandas df grouped on published_date month and
year, on which 'sum' is applied on num_occurrences and then normalized to
get a percentage.
             docs_df_unique, a Pandas df grouped on published_date month
and year, on which 'count' is applied on num_occurrences and then
normalized to get a percentage."""
    # Get a list of dictinoaries by parsing the JSON results for the search
query
    docs = search_solr_parse_json(query, "nounphrases", "phrase")
    if docs == []:
        # No data found
        return None, None
    # Create a pandas dataframe out of the result
    docs_df = dataframe_from_solr_results(docs)
```

```
    # Group by published_date, and calculate sum and count of
num_occurrences.
    #These correspond to total_occurrences of a phrase for a date, and
unique
    # occurrences of a phrase for a date.
    docs_df_total, docs_df_unique = calculate_aggregates(docs_df)
    docs_df_total, docs_df_unique =
get_percentage_aggregates(docs_df_total, docs_df_unique)
    return docs_df_total, docs_df_unique
```

Step 4:

Now, we need to define the Dash functions and the actual Plotly graphs. The process is described only for the **'Percentage of occurrences of chosen noun phrase(s) per Month'** graph. The **'Percentage of papers containing chosen noun phrase(s) per Month'** is very similar and won't be covered in this document. Like the other functions, the comments in the program are very detailed.

The callback indicates that the wrapped (show_graph_total) function is called whenever the state of the input field is changed by clicking the submit button. This is changed whenever the n_clicks property of the Html submit button element has been changed.

It returns the graph in its output div (and the show_graph_unique function which is not shown here similarly returns its graph in its output div).

The code below allows the user to enter multiple search phrases separated by commas. These are split and the get_aggregated_data function is called on each of these noun phrases after standardizing them.

Any phrase which does not return a result from Solr (get_aggregated_data returns None) is added to a 'notfound_list'. A message is displayed in the output Div listing the elements not found (the message is built in another function).

Plotly expects a single layout and a data element together in a dictionary. The layout consists of things like the axes labels, title, colour and type of graph (bar in our case). The ' data' element consists of go.Bar elements for each of the user's phrase queries.

These are rendered in the webpage using Dash core components's 'Graph' function. Note that the list of elements not found is also displayed if any elements are not found. This list is displayed in the output div for the first graph only.

Note that if none of the phrases are found, no graph is rendered, and only the list of phrases which haven't been found are output on the webpage.

The callback function is given below. Pay attention to the Layout. The x-axis ticks should start from April 2007, and subsequent ticks should be 2 months apart. The range property makes sure there is a tick for April (and Dec 2017) even if there is no data for those months. The values should be slightly before/after the actual month we want on the label, otherwise Plotly makes a half-bar.

```
@app.callback(Output('output_total', 'children'),
              [Input('submit-button', 'n_clicks')],
              [State('npinput1-state', 'value')])
def show_graph_total(n_clicks, input_box):
    """ Wrapped function which takes user input in a text box, returns a graph
```

```
    if the query produces a hit in Solr, returns an error message otherwise.
    ARGUMENTS: n_clicks: a parameter of the HTML button which indicates it has
               been clicked
               input_box: the content of the text box in which the  user has
               entered a comma-separated search query.
    RETURNS: 1 graph (total occurrences) of all terms which have results from
             Solr, error messages of all terms which don't have results from
Solr."""

    # Store the layout with the appropriate title and y axis labels for the graph
    # xticks should start from April 2007, and subsequent ticks should be 2 months
apart. The range property
    # makes sure there is a tick for April (and Dec 2017) even if there is no data
for those months. The values
    # should be slightly before/after
    layout_total = go.Layout(
                    title = 'Percentage of occurrences of chosen noun phrase(s)
per Month',
                    xaxis = {'title': 'Publication date', 'tickformat': '%b %y',
'tick0': '2007-04-30',
                             'dtick': 'M2', 'range': ['2007-03-25', '2018-01-
25']},
                    yaxis = {'title': 'Percentage of phrase occurrences',
'ticksuffix': '%'},
                    plot_bgcolor = colours['background'],
                    paper_bgcolor = colours['background'],
                    barmode = 'stack',
                    hovermode = 'closest',
                    font= {
                            'color': colours['text']
                          },
                    showlegend=True
                    )

    if input_box != '':
        # Get the input data: both freq_df dfs will have index= published_date,
        # columns = percentage_occurrences total.
        input_list = input_box.lower().split(',')
        data_list_total = []
        notfound_list = []
        for input_val in input_list:
            # Make sure to strip input_val, otherwise if the user enters a
            # space after the comma in the query, this space will get sent
            # to Solr.
            freq_df_total, freq_df_unique = get_aggregated_data(input_val.strip())
            if freq_df_total is not None:
                # Plot the graphs, published_date (index) goes on the x-axis,
                # and percentage_occurrences total goes on the y-axis.
                data_list_total.append(go.Bar(
                                    x = freq_df_total.index,
                                    y = freq_df_total.percentage_occurrences,
                                    text = input_val.strip().capitalize(),
                                    opacity = 0.7,
                                    name = input_val.strip().capitalize()
                                    ))

            else:
                # Term not found, append it to the not found list and go to the
                # next term.
```

```
                notfound_list.append(input_val)

        if data_list_total == []:
            if notfound_list != []:
                # Append the error message for the terms not found in the
                # Solr index
                return not_found_message(notfound_list)

            # One or more of the Solr queries returned a result
        else:
            #graph_total_terms = {'data': data_list_total, 'layout': layout_total}
            graph_total_terms = dict(data=data_list_total, layout=layout_total)
            if notfound_list != []:
                terms_not_found = not_found_message(notfound_list)
                #return terms_not_found, html.Br(),
                return terms_not_found, dcc.Graph(id='totalfreq', figure=
graph_total_terms)

            return html.Br(), dcc.Graph(id='totalfreq', figure= graph_total_terms)
```

The function to build the not-found message for all the terms in the not found list is pretty straightforward.

```
def not_found_message(notfound_list):
    """ Takes a list of elements not found in the Solr index and produces
    an error message for the whole lot of them together, along with
suitable
    styling (in an <h3> tag).
    ARGUMENTS: notfound_list: list of user's search terms which are not
found
               in the Solr index
    RETURNS: a html h5 message with a message listing the terms not
found"""
    notfound_list = ['"' + term.strip().capitalize() + '"'
                     for term in notfound_list]
    notfound = ','.join(notfound_list)
    return html.H5('Noun phrases not found: {}.'.format(notfound),
            style={'color': colours['text']}
            )
```

Step 5: The above process is repeated for the second graph.

Step 6:

The main function just calls flask's run_server method (dash is a flask app). 0.0.0.0 is used so that the webpage can be accessed from other systems on the network.

```
if __name__ == '__main__':
    app.run_server(host='0.0.0.0')
```

EXAMPLE:

The program is executed as follows:

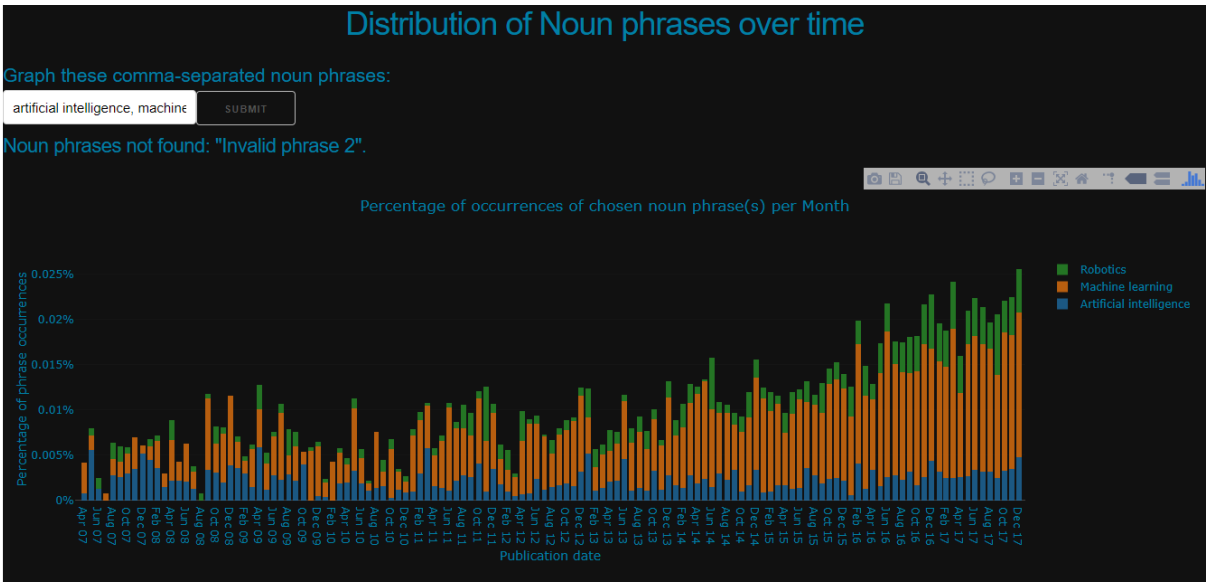
python3 nounphrase_visualization.py

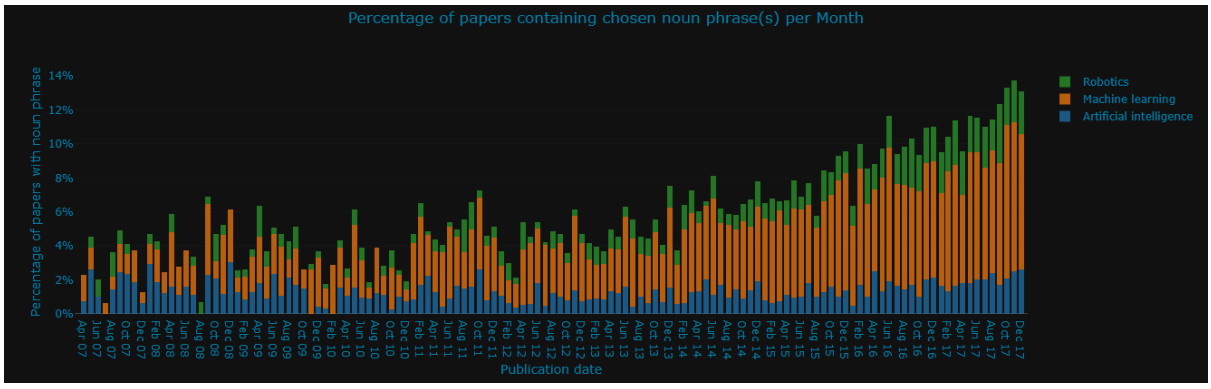
The development server runs on port 8050, so the following IP address (for the 'Kilda' machine which hosts the code) needs to be typed in the browser with port 8050.

http://132.230.150.9:8050/

The search query in this example is: **artificial intelligence, invalid phrase1, invalid phrase2, machine learning.**

The result is shown below (the 2 graphs are shown in 2 screenshots). Both graphs are returned simultaneously in the same webpage.





ADDENDUM: Yearly Noun Phrase Visualization

Created Python module: nounphrase_visualization_yearly.py, month_to_year_json.py

Prerequisites: Same as nounphrase_visualization_monthly.py

In yearly noun phrase visualization, the grouping is done by year instead of month in the main program

The grouping is by year, as mentioned.

To calculate the percentages, we need to use a new JSON file created from the monthly file. This JSON file contains yearly phrase count and document count. This code to convert the monthly json file to a yearly json file (with summing up) is shown below.

```
with open('phrases_and_docs_monthly.json', 'r') as file:
        json_array= json.load(file)
    # json_array is a list of 2 dicts (1st -> phrase freq, 2nd -> doc_freq)
    monthly_phrases_total = json_array[0]
    monthly_docs_total = json_array[1]
    # Create a list of years to be used to add values corresponding to all keys in
a year
    years = ['2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014',
'2015', '2016', '2017']
    yearly_phrases_total = dict()
    yearly_docs_total = dict()
    # Create
    for year in years:
        yearly_phrases_total[year] = sum([freq for month, freq in
monthly_phrases_total.items() if month.startswith(year)])
        yearly_docs_total[year] = sum([freq for month, freq in
monthly_docs_total.items() if month.startswith(year)])

    list_for_json = [yearly_phrases_total, yearly_docs_total]
    # Dump the list to a json file (as a json array with 2 json objects: 1st ->
yearly phrase freq, 2nd -> yearly doc freq)
    with open('phrases_and_docs_yearly.json', 'w') as outputfile:
        json.dump(list_for_json, outputfile)
```

The calculate_aggregates and get_percentage_aggregates functions also change a bit.

E.g. calculate_aggregates function has this line changed, among other things:

```
docs_df_total = docs_df.groupby(pd.Grouper(freq='1Y')).sum()
```

And the get_percentage_aggregates is changed as shown below.

```
    with open('phrases_and_docs_yearly.json', 'r') as file:
        json_array= json.load(file)
    # json_array is a list of 2 dicts.
    yearly_phrases_total = json_array[0]
    yearly_docs_total = json_array[1]
    # For each of the dataframes, create a year column.This is a string and
matches the value from the json file.
    # Create year column as a period object with frequency = every year
    docs_df_total['year'] = docs_df_total.index.to_period('Y')
    # Convert the period object to a string
    docs_df_total.year =  docs_df_total.year.astype('str')
    # Create a new column which uses the value in the year string column as a key
in the yearly_phrases_total
    # dict, and gets the corresponding value. The no. of occurrences is divided by
this number. The na_action is not strictly necessary, it is just a precaution
which inserts NaN if a key (year) is not found. Finally, NaNs are
```

```
    # produced if the dict value has a 0 (divide by 0). These NaNs are replaced by
0. * 100 because the final result is in %.
    docs_df_total['percentage_occurrences'] = (100 * docs_df_total.num_occurrences
/ docs_df_total['year']
        .map(yearly_phrases_total, na_action=None)).fillna(0)
```

The graph layout also changes, as below.

```
    # Store the layout with the appropriate title and y axis labels for the graph
    # xticks should start from 2007, and subsequent ticks should be 12 months (1
year) apart. The range
    # property makes sure there is a tick for 2007 and 2017 even if there is no
data for those years.
    # The values should be slightly before/after the first/last bar
    layout_total = go.Layout(
                    title = 'Percentage of occurrences of chosen noun phrase(s)
per Year',
                    xaxis = {'title': 'Publication year', 'tickformat': '%Y',
'tick0': '2007-12-31',
                             'dtick': 'M12', 'range': ['2007-07-01', '2018-07-
01']},
                    yaxis = {'title': 'Percentage of phrase occurrences',
'ticksuffix': '%'},
                    plot_bgcolor = colours['background'],
                    paper_bgcolor = colours['background'],
                    barmode = 'stack',
                    hovermode = 'closest',
                    font= {
                            'color': colours['text']
                          },
                    showlegend=True
                    )
```

Similar changes are made for the documents graph as well, where the aggregation has to be done per year.

Finally, the graphs themselves are shown in the following graphs (on the next page). The query is the same as for the monthly noun phrase visualization.

Distribution of Noun phrases over time

Graph these comma-separated noun phrases:

artificial intelligence, machine | SUBMIT

Noun phrases not found: "Invalid phrase 2".