Winter 2018 LIGN 199 Writeup

Xiaoqing (Tom) Li

March 24, 2018

In Winter Quarter of 2018, I was enrolled in LIGN 199, a special independent undergraduate research course. I assisted Matthew Zaslansky, a graduate student at UCSD, with research in Azerbaijani, chiefly data compilation from an online corpus, SketchEngine, and processing of that data, in Python. All code from the project can be found in my GitHub repository, https://github.com/variablehair/199wi18. The course is graded as P/NP and is worth four credits.

1 Introduction

A relatively small number of external packages and modules were used in this project; they will be briefly introduced here.

1.1 `json`

The `json` package is used widely throughout the project as one of the two primary data management formats. Primarily, it is used to save and load data to disk in a machine-friendly format.

```
r = s.get(url, params = attrs)
curr_res_dict = r.json()
...
with open('data/temp/' + query + '100.json', mode='w',
↪  encoding='utf=8') as f:
    json.dump(curr_results, f)
```

In the above snippet from `data_fetch.py`, data is retrieved from SketchEngine in JSON format, and after processing, is then stored in a `.json` file.

```
with open('data/dative_pruned.json') as f:
    j_dat = json.load(f)

for token in j_dat:
    ...
```

In the above snippet from `xlsx_writer_task2.py`, stored JSON data is loaded into the program. This approach, while simple, ran into a problem of magnitude; specifically, `json.load` was found to have significant overhead on memory (RAM) which seemed to scale exponentially with file size, rendering this approach untenable when the data set became large, as ours did.

1.2 sqlite3

As a solution to this problem, the dataset was transposed to a SQLite database using the `sqlite3` package. `sqlite3` was chosen because of its minimal setup and upkeep, making it ideal for our task. Its primary drawback is its relatively slower speed, but as our total and average throughput of database access was extremely low, this was not a concern.

```
    conn = sqlite3.connect('data/data100.db')
    ...
    def process_raw_json(filename, tokenname, dbconn):
        c = dbconn.cursor()
    try:
        c.execute('''CREATE TABLE {} (token TEXT, left TEXT,
        ↪  right TEXT)'''.format(tokenname))
        ...
    for string_dict in string_list:
        query = '''INSERT INTO {0} VALUES
        ↪  ('{1}','{2}','{3}')'''.format(tokenname,
        ↪  result_dict['Kwic'][0]['str'],
        ↪  left_str.replace("\'", "\'\'"),
        ↪  right_str.replace("\'", "\'\'"))
        try:
            c.execute(query)
        ...
```

In the above snippet from `compile_data100.py`, tables are created in our database and populated with data using database connection and cursor objects.

```
    c.execute('''SELECT token, left, right FROM
    ↪  {0}'''.format(token))

    while True:
        r = c.fetchone()
        if r is None:
            break
        word, left, right = r
        ...
```

In the above snippet from `counter_task2_dict.py`, a query which fetches

every row from a particular table is executed. By using `c.fetchone()`, we only load a single row of data at a time into memory, bypassing the JSON problem (wherein the entire dataset had to be loaded into memory).

1.3 `xlsxwriter`

The `xlsxwriter` package is used several times throughout the project to generate human-readable `.xlsx` files once data had been processed. It was chosen from several options because it was the most lightweight due to lacking the functionality to open existing `.xlsx` files. However, this functionality was not required in this project.

```
workbook_dat =
↪  xlsxwriter.Workbook('dict_pruned_dative.xlsx')
...
for token in j_dat:
    worksheet = workbook_dat.add_worksheet(token)
    row = 0
    ...
    for tup in tup_results:
        worksheet.write(row, 0, word)
        worksheet.write(row, 1, tup[0])
        worksheet.write(row, 2, tup[1])
    row += 1
workbook_dat.close()
```

In the above snippet from `xlsx_writer_task2.py`, `xlsxwriter` is used to create a file (`Workbook`), sheets (`worksheet`s), and then write particular results iteratively to each row.

1.4 `string`

The `string` package is used to access various built-in constants, such as a list of all the punctuation.

```
import string
...
    word = word.strip(string.punctuation + string.digits +
    ↪  '')
    ...
```

In the above snippet from `simple_get_dict_entry.py`, `string.punctuation` and `string.digits` are used as arguments to `strip()` in order to remove trailing punctuation marks and numerals from dictionary entries as part of the process to convert them to clean, machine-readable format.

1.5 `re`

The regular expression package `re` is used to perform more advanced trimming operations on the dictionary entries.

```
try:
    if line[0].isupper():
    curr = re.sub(r'\([^()]*\)', '', line[0])
    ...
```

In the above snippet from `syntactic_category_fetch.py`, a regular expression is used to remove all strings between parentheses within words, as such strings were commonly found in the dictionary.

1.6 `requests` and `urllib.parse`

The `requests` package and `urllib.parse` module are used only in the data fetching process to interface with SketchEngine's API.

```
s = requests.Session()
s.auth = (tom_username, tom_password)
s.get(login_url)
url = base_url + 'view'
...
r = s.post(login_url,data=logindata)
encoded_attrs=urllib.parse.urlencode(attrs)
r = s.get(url, params = attrs)
```

`requests` is used in the above snippet from `data_fetch.py` to create an interface with SketchEngine via a `Session` object, which sends and receives data with `post` and `get`. `urllib.parse` is used to encode the data sent to the server into the appropriate format.

1.7 `time`

The `time` package is used only in the data fetching process to delay execution of the code in order to comply with SketchEngine's rate limits.

```
while pagenum < MAX_SAMPLES:
pagenum += 1
attrs['fromp'] = pagenum
r = s.get(url, params=attrs)
curr_res_dict = r.json()
...
time.sleep(10)
if pagenum % 25 == 0:
time.sleep(60)
```

In the above snippet from `data_fetch.py`, `time.sleep` is used to pause execution of the loop on every iteration for 10 seconds with an additional pause on every 25th iteration for 60 seconds. This was found to be within the limits allowed by SketchEngine despite being less than their prescribed pause (44 seconds per query), which is prohibitively long for our task.

1.8 `os` and `ast`

`os` and `ast` used only in `compile_data100.py` and are used to work with the filesystem and raw data respectively

```
datapath = 'data/temp/'
datadir = os.fsencode(datapath)
...
for file in os.listdir(datadir):
    filename = os.fsdecode(file)
    tokenname = filename[0:-8]
    process_raw_json(datapath+filename, tokenname, conn)
```

In the above snippet from `compile_data100.py`, `os.fsencode` and `os.fsdecode` are used to allow Python to iterate through a folder containing raw data in JSON format.

```
split_string = '{\"rightsize'
string_list = [split_string + s[:-2] for s in
↪  raw_string.split(split_string) if len(s) >= 2]
for string_dict in string_list:
    try:
        result_dict = ast.literal_eval(string_dict)
    except SyntaxError:
        result_dict = ast.literal_eval(string_dict + '}')
```

In the above (somewhat hacky) snippet from `compile_data100.py`, `ast.literal_eval()` is used to convert a string to a Python data structure, in this case a dictionary. This approach was necessary because the files in question were too large to load directly despite being in JSON format, so they had to be loaded as strings before being converted.

1.9 `operator.itemgetter()`

The module `operator.itemgetter()` was used as an efficient alternative to the built-in `dict.get()` for efficiency reasons. Though it probably would not have made a tangible difference, our data sets could sometimes be very large, and so `itemgetter()` was used as a safeguard.

```
    results_list = sorted(found_words.items(),
    ↪  key=itemgetter(1), reverse=True)
```

`itemgetter()` was used exclusively in `sorted()`, which produced a sorted list of tuples from a dictionary.

2 Structure and non-code files

A number of non-Python files (e.g. `.txt`, etc.) are found in the repository. They, along with the structure of the repository, will be very briefly detailed here.

2.1 Structure

The repository's structure is very simple. Most files are found at the top level. The directory `azer_dictionary_cleanup` contains files relevant to our work converting and working with an Azerbaijani dictionary, and the directory `data` contains data in various forms. However, note that the data directory is not actually available in the repository due to size constraints (e.g. the database is 267MB) despite being frequently used in the code. This means that trying to run the code directly after cloning the repository will result in errors; the data can, of course, be provided upon request. Lastly, unless otherwise noted, all files use UTF-8 encoding.

2.2 `dict.txt`, `dict_entries.txt`, and `dict_entries_noparens.txt`

These three files contain the Azerbaijani dictionary and information from it in text format. `dict.txt` contains a plaintext version of the dictionary itself. `dict_entries.txt` contains a roughly parsed and unpruned list of entries from the dictionary, and `dict_entries_noparens.txt` contains the same list, but with word-internal parenthesized blocks removed.

2.3 `task1clean.txt`

This probably poorly named file contains a list of all query strings (e.g. "dir", "ar", etc.) used in our project. It is called 'clean' because it contains only the raw search strings without either artifacts from when this list was sent (e.g. preceding hyphens) or when these strings are used as actual search queries in the corpus (i.e. asterisks/regex formatting). The results from these queries comprised all of the data used, and this file is used in several places, e.g. to name the tables in the database.

3 Individual Python script overviews

In this section, the individual Python modules will be described in rough chronological order. Note that some will be deprecated (and marked as such), mostly due to the change from JSON to SQLite.

3.1 data_fetch.py

`data_fetch.py` interfaces with the SketchEngine API, sends queries, receives data, and then saves it to disk.

```
from login import tom_username, tom_password
import requests
import json
import urllib.parse
import time

MAX_SAMPLES = 100

root_url = 'https://the.sketchengine.co.uk'
base_url='%s/bonito/run.cgi/' % root_url

login_url = 'https://the.sketchengine.co.uk/login/'
logindata = {'username' : tom_username, 'password' :
↪  tom_password,'submit' : 'ok'}

qlist = []
with open('task1clean.txt', encoding='utf-8') as queries:
    for line in queries:
        qlist.append(line.replace('\n', ''))

s = requests.Session()
s.auth = (tom_username, tom_password)
s.get(login_url)
url = base_url + 'view'

api_log = open('api_log.txt', encoding='utf-8', mode='w')
```

Some general definitions and imports, part of which comes directly from SketchEngine's example script. `MAX_SAMPLES` is probably poorly named, but is the maximum number of pages per query; there are 1,000 results per page, so this sets the limit for maximum number of results per query at 100,000. `qlist` is a list of queries that we are interested in. `api_log` is a log file for debugging purposes. The remaining variables are various API-related operators, e.g. `logindata` is imported and then used to authorize the session, allowing us to make further requests.

```
for query in qlist:
    pagenum = 1
    fquery = str('q[word=".*' + query + '.*"]')
    attrs = dict(corpname = 'preloaded/turkic_az', q =
    ↪  fquery, pagesize = '1000', format = 'json',
    ↪  fromp=pagenum, r='r1000')
    curr_results = list()

    r = s.post(login_url,data=logindata)

    encoded_attrs=urllib.parse.urlencode(attrs)

    r = s.get(url, params = attrs)
    curr_res_dict = r.json()

    try:
        e = (curr_res_dict['error'])
        api_log.write('ERROR on ' + str(query) + ':\t' +
        ↪  str(e) + '\n')
        continue
    except KeyError:
        pass

    curr_results = curr_res_dict['Lines']
    try:
        total_pages = curr_res_dict['numofpages']
    except KeyError:
        total_pages = 1
    api_log.write(query + ' page ' + str(pagenum) + 'done,
    ↪  len = ' + str(len(curr_results)) + '\n')
```

This section sets up the `for` loop which encapsulates the rest of the file (keep in mind that the following sections are within the loop). A request is first constructed in the `attrs` object; note that pagesize is 1000, the maximum allowed. `r` is supposed to return a random sample of results according to the documentation, but it seems to not do this (or perhaps is incompatible with another option, but is not documented). `fromp` is the page that is returned, starting with 1, and in our case increasing to a maximum of 100.

This request is then sent to the corpus with `requests.get()`, and the result is converted to a dictionary using the `json` library. The first thing that is checked is whether there is an error, either a rate limit or an empty result; if so, we move on to the next iteration (i.e. the next query) with `continue`. Finally, if there is no error, we save the current page of results (the data itself is found in `Lines`) and then count the total number of pages.

```
        if total_pages > 1:
            while pagenum < MAX_SAMPLES:
                pagenum += 1
                attrs['fromp'] = pagenum
                r = s.get(url, params=attrs)
                curr_res_dict = r.json()
                curr_results = curr_results +
                ↪  curr_res_dict['Lines']
                api_log.write(query + ' page ' + str(pagenum) +
                ↪  'done, len = ' + str(len(curr_results)) +
                ↪  '\n')
                time.sleep(10)
                if pagenum % 25 == 0:
                time.sleep(60)
                print(str(pagenum))

        with open('data/temp/' + query + '100.json', mode='w',
        ↪  encoding='utf=8') as f:
        json.dump(curr_results, f)
        curr_results = []

        time.sleep(120)

    api_log.close()
```

In this section, we essentially repeat the loop if there are multiple pages of results, up to our defined maximum. Note that there are multiple `time.sleep()` calls in order to avoid rate limits from SketchEngine. Finally, after all the data has been compiled, it is unceremoniously dumped to a JSON file in its raw form.

3.2 `counter_task1.py`

Deprecated.

This module was created for the first task, which was simply an ordered token count of words which contained query strings. It has not been updated with the new database. It also does not contain optimizations and improvements found in later scripts.

```
import json
import xlsxwriter

f = open('data/task1.json')
d = json.load(f)
k = list(d.keys())
```

```
    workbook = xlsxwriter.Workbook('data.xlsx')

    for key in k:
        try:
            l = d[key]['Lines']
        except KeyError:
            continue

    results = dict()

    for r in l:
        try:
            word = r['Kwic'][0]['str']
            results[word] = results.get(word, 0) + 1
        except KeyError:
            continue

    worksheet = workbook.add_worksheet(key)

    row = 0

    for result in results.keys():
        row += 1
        worksheet.write(row, 0, result)
        worksheet.write(row, 1, results[result])

    workbook.close()
```

This module tries to access the `Lines` object of the result JSON file, which contains a page of results. After that, it tries to access `r['Kwic'][0]['str']`, which contains the word which contains the query string. A results dictionary then counts the number of occurrences of such words.

3.3 char_replace.py

This module replaces characters from the custom encoding found in the Azerbaijani dictionary to the more common UTF-8.

```
    with open("in.txt", encoding="utf-8") as infile, open
    ↪  ("out.txt", mode="w", encoding="utf-8") as outfile:
        for line in infile:
            for char in line:
                try:
                    out_char = char_dict[char]
                    outfile.write(out_char)
```

```
                except KeyError:
                    outfile.write(char)
```

Not included due to space is the dictionary which maps the replacement characters. This module iterates through the file and simply replaces all characters found in the dict; if the character is not found, it simply writes it with no changes.

3.4 simple_get_dict_entry.py

This module attempts to extract a list of dictionary entries from the converted dictionary in text form.

```
import string

with open("dict.txt", encoding="utf-8") as infile,
↪  open("dict_entries.txt", encoding="utf-8", mode="w") as
↪  outfile:
    for line in infile:
        try:
            word = line.split(maxsplit=1)[0]
            if word.isupper():
                word = word.strip(string.punctuation +
                ↪  string.digits + '')
                if len(word) >= 2:
                    outfile.write(word + "\n")
        except IndexError:
            continue
```

Each line is split at whitespace, and the first word is examined. If it is entirely in uppercase, and longer than 2 characters after punctuation or numerals have been removed, then it is added to the list of entries. Finer pruning was certainly possible but perhaps not needed for this task.

3.5 compile_data100.py

This module takes the JSON data from `data_fetch.py` and puts it into a SQLite database. Because the JSON data already existed, it was written to work with that specifically; a much better approach for any future implementation would be to have the data fetch module work directly with the database, eliminating this middle man.

```
def process_raw_json(filename, tokenname, dbconn):
    c = dbconn.cursor()
    try:
```

```
            c.execute('''CREATE TABLE {} (token TEXT, left TEXT,
            ↪  right TEXT)'''.format(tokenname))
        except sqlite3.OperationalError:
            print(tokenname)
            return
        with open(filename, encoding='utf-8') as data:
            raw_string = data.readline()
            split_string = '{\"rightsize'
            string_list = [split_string + s[:-2] for s in
            ↪  raw_string.split(split_string) if len(s) >= 2]
            for string_dict in string_list:
                try:
                    result_dict = ast.literal_eval(string_dict)
                except SyntaxError:
                    result_dict = ast.literal_eval(string_dict +
                    ↪  '}')
                left_str = [ldict['str'].split(' ') for ldict in
                ↪  result_dict['Left']]
                left_str = [string for sublist in left_str for
                ↪  string in sublist]
                left_str = ' '.join(left_str)
                right_str = [rdict['str'].split(' ') for rdict
                ↪  in result_dict['Right']]
                right_str = [string for sublist in right_str for
                ↪  string in sublist]
                right_str = ' '.join(right_str)
                query = '''INSERT INTO {0} VALUES
                ↪  ('{1}','{2}','{3}')'''.format(tokenname,
                ↪  result_dict['Kwic'][0]['str'],
                ↪  left_str.replace("\'", "\'\'"),
                ↪  right_str.replace("\'", "\'\'"))
                try:
                    c.execute(query)
                except sqlite3.OperationalError:
                    print(query)
        conn.commit()
        print(tokenname)
```

This function makes up the bulk of the module. It takes a file, a token, and a database connection. It attempts to create a table with the token (e.g. 'ar'), and then reads the entire file as a string (because it cannot be loaded as an object due to size). It then splits this raw string at `'\{rightsize'`, which precedes each unique entry in the data. It then evaluates each individual entry as a literal Python object rather than a string before concatenating them (as they are in one or two layers of lists) into a string, which is finally inserted into

the database.

```
import os
import json
import sqlite3
import ast

datapath = 'data/temp/'
datadir = os.fsencode(datapath)
conn = sqlite3.connect('data/data100.db')

def process_raw_json(filename, tokenname, dbconn):
    ...

for file in os.listdir(datadir):
    filename = os.fsdecode(file)
    tokenname = filename[0:-8]
    process_raw_json(datapath+filename, tokenname, conn)

conn.close()
```

The remainder of the file simply iterates through the folder containing all the JSON objects and calls the previously defined function on each of them.

3.6 counter_task2_dict.py

This module contains task 2, which attempted to extract dative and accusative marked nouns from the sentence.

```
import json
import sqlite3

EXCLUDE_LEFT = ['mli', 'mal', 'mk', 'maq']
DAT_CASE = ['a', '']
ACC_CASE = ['i', 'u', '', '']
EXCLUDE_PREV_WORD = ['in', 'un', 'n', 'n']
END_OF_SENTENCE = ['.', '?', '!', '\n', '<', '>']

conn = sqlite3.connect('data/data100.db')
c = conn.cursor()

with open('dict_entries_noparens.txt', encoding='utf-8') as
↪   f:
    DICT_ENTRIES = set([string[:-1] for string in
    ↪   f.readlines()])
```

```
    tokens = conn.execute('''SELECT name FROM sqlite_master
    ↪  WHERE type="table"''')
    tokens = [tup[0] for tup in tokens]

    pruned_dict_entries = open('pruned_dict_entries.txt',
    ↪  encoding='utf-8', mode='w')
    all_results_pruned = open('all_results_pruned.txt',
    ↪  encoding='utf-8', mode='w')
    no_results_found = open('no_results_found.txt',
    ↪  encoding='utf-8', mode='w')
```

Some definitions, such as case markers, are given. Pruning-related constants, such as the a set of dictionary entries, are created, and a list of all tables to be iterated through are also created. Finally, some log files to track pruning are created.

```
    def EOS_prune(word_list, kept_words, position):
        if position == len(word_list) - 1:
            return word_list
        elif len(word_list[position]) > 0 and
        ↪  word_list[position][-1] in END_OF_SENTENCE:
            return kept_words
        else:
            kept_words.append(word_list[position])
            return EOS_prune(word_list, kept_words, position+1)
```

This function takes a list of words and returns the same list of words with all words after the sentence boundary removed by recursively iterating until a end of sentence marker, such as a period or an HTML tag (all tags found in a quick sample were paragraph break tags, so we assumed they marked a sentence boundary).

```
    def pull_cased_nouns(word_list, case_markers, position,
    ↪  pulled_words):
        if position >= len(word_list) - 1:
            return pulled_words
        curr_word = word_list[position]
        if len(curr_word) >= 3 and curr_word[-1] in
        ↪  case_markers:
            if curr_word.upper() in DICT_ENTRIES:
                pruned_dict_entries.write(word + '\t\t\t\t\t' +
                ↪  curr_word + '\n')
                return pull_cased_nouns(word_list, case_markers,
                ↪  position+1, pulled_words)
```

```
            if curr_word[-2] == 'n':
                if position >= 1 and word_list[position-1][-2:]
                ↪  in EXCLUDE_PREV_WORD:
                return pull_cased_nouns(word_list, case_markers,
                ↪  position+1, pulled_words)
            elif position == 0:
                return pull_cased_nouns(word_list, case_markers,
                ↪  position+1, pulled_words)

            if curr_word[-2:] in ['da', 'd']:
                return pull_cased_nouns(word_list, case_markers,
                ↪  position+1, pulled_words)

            pulled_words.append(curr_word)
        return pull_cased_nouns(word_list, case_markers,
        ↪  position+1, pulled_words)
```

This function takes a list of word and extracts nouns marked by the provided case markings according to our pruning rules, and also keeps track of the previous word for this purpose. Pruning rules include being a dictionary entry and the marking of the previous word.

```
    results_dat = dict()
    results_acc = dict()

    for token in tokens:
    token_dat_results = dict()
    token_acc_results = dict()

    c.execute('''SELECT token, left, right FROM
    ↪  {0}'''.format(token))

    while True:
        r = c.fetchone()
        if r is None:
            break
        word, left, right = r

        try:
            if word.endswith(token):
                #exclude word-final appearance
                continue

            left = left.split(' ')
            right = right.split(' ')
```

```
            for left_word in left:
                if any(x in left_word for x in EXCLUDE_LEFT):
                    #exclude specific left-occuring words
                    raise ValueError

            left.reverse()
            #remove words outside of sentence
            left = EOS_prune(left, [], 0)
            left.reverse()
            right = EOS_prune(right, [], 0)

            pulled_dat_list = pull_cased_nouns(left, DAT_CASE,
            ↪  0, []) + pull_cased_nouns(right, DAT_CASE, 0,
            ↪  [])
            pulled_acc_list = pull_cased_nouns(left, ACC_CASE,
            ↪  0, []) + pull_cased_nouns(right, ACC_CASE, 0,
            ↪  [])

            curr_dat_results = token_dat_results.get(word, {})
            curr_acc_results = token_acc_results.get(word, {})

            for pulled_dat in pulled_dat_list:
                curr_dat_results[pulled_dat] =
                ↪  curr_dat_results.get(pulled_dat, 0) + 1
            token_dat_results[word] = curr_dat_results

            for pulled_acc in pulled_acc_list:
                curr_acc_results[pulled_acc] =
                ↪  curr_acc_results.get(pulled_acc, 0) + 1
            token_acc_results[word] = curr_acc_results

        except ValueError:
            continue

    results_dat[token] = token_dat_results
    results_acc[token] = token_acc_results

    with open('data/dative_pruned.json', mode='w') as f:
        json.dump(results_dat, f)

    with open('data/accusative_pruned.json', mode='w') as f:
        json.dump(results_acc, f)

    pruned_dict_entries.close()
```

```
    all_results_pruned.close()
    no_results_found.close()
```

This remainder of the module iterates through every table in the database, and for each one, tests one row at a time, extracting appropriately marked nouns. Some basic pruning, such as word-final appearance of the verb marking, is done here. Note that the left-occurring word pruning is incorrect and should be changed if we return to this task in the future. The left and right strings are then passed through the two previously defined functions – note that the left string is first reversed, as we want to remove the words outside rather than inside of the sentence. The dative and accusative results are separately recorded before being saved in JSON form.

3.7 xlsx_writer_task2.py

This module takes the JSON files generated by the previous module and writes them to human-readable xlsx format. The reason for this two-step process is that it is much more difficult to work directly with xlsx files than with JSON data, and so it is useful to keep the intermediate JSON representation around.

```
    import xlsxwriter
    import json
    from operator import itemgetter

    workbook_dat =
    ↪  xlsxwriter.Workbook('dict_pruned_dative.xlsx')
    with open('data/dative_pruned.json') as f:
        j_dat = json.load(f)

    for token in j_dat:
        worksheet = workbook_dat.add_worksheet(token)
        row = 0
        word_dict = j_dat[token]
        for word in word_dict:
            tup_results = sorted(word_dict[word].items(),
            ↪  key=itemgetter(1), reverse=True)
            for tup in tup_results:
            worksheet.write(row, 0, word)
            worksheet.write(row, 1, tup[0])
            worksheet.write(row, 2, tup[1])
            row += 1
    workbook_dat.close()
```

This module contains the same process for both dative and accusative case (and really should use a function instead of copy-pasting code), so only one of

the two is provided due to space. The JSON file is loaded and a sheet is created for each original search query; then, the module iterates through the verbs, sorts their results, then writes them in descending order.

3.8 xlsx_pruned_dict_writer.py

This module should really be called 'dict_pruned_writer', but oh well. This module simply takes the text file of pruned entries, concatenates them by number of occurences, and writes them into an xlsx file for readability.

```
import xlsxwriter

with open('pruned_dict_entries.txt', encoding='utf-8') as f:
    lines = f.readlines()
    tuple_list = []
    for line in lines:
    sl = line.split('\t')
    tuple_list.append((sl[0], sl[-1]))

results = {}
for tup in tuple_list:
    tup_results = results.get(tup[0], {})
    tup_results[tup[1]] = tup_results.get(tup[1], 0) + 1
    results[tup[0]] = tup_results

workbook = xlsxwriter.Workbook('pruned_dict_entries.xlsx')
worksheet = workbook.add_worksheet('Pruned Entries')

row = 0
for r in results:
    curr_dict = results[r]
    for key in curr_dict:
        worksheet.write(row, 0, r)
        worksheet.write(row, 1, key)
        worksheet.write(row, 2, curr_dict[key])
        row += 1

workbook.close()
```

The text file of pruned entries is read in. Because of the formatting of that file, we split each line at tabs, resulting in a tuple, from which we take the first and last members. Then, we simply group results and count occurrences before writing the results to an xlsx file.

3.9 task3_hapax.py

This module counts the number of *hapax legomena* occurring within our data set for each causative.

```
import sqlite3
import xlsxwriter
from operator import itemgetter

with open('task1clean.txt', encoding='utf-8') as f:
    token_list = f.readlines()

conn = sqlite3.connect('data/data100.db')

workbook = xlsxwriter.Workbook('task3_hapax.xlsx')
```

As with previously covered modules, some basic definitions and imports, such as a database connection, are created.

```
for token in token_list:
    c = conn.cursor()
    query = '''SELECT token FROM {0}'''.format(token)
    c.execute(query)

    total = 0
    hapax = 0
    found_words = dict()

    while True:
        row = c.fetchone()
        if row == None:
            break
        found_words[row[0]] = found_words.get(row[0], 0) + 1

    for word in found_words:
        num = found_words[word]
        total += num
        if num == 1:
            hapax += 1

    results_list = sorted(found_words.items(),
    ↪  key=itemgetter(1), reverse=True)
```

The module then iterates through all tables in the database, and fetches the verb (‘word’) from each row, keeping count of the number of times each verb occurs. Afterward, it iterates through these results and calculates both the total number of verbs as well as the number of hapaxes.

```
    worksheet = workbook.add_worksheet(token)

    row = 0
    worksheet.write(row, 0, 'Total hapax')
    worksheet.write(row, 1, 'Total tokens')

    row += 1
    worksheet.write(row, 0, hapax)
    worksheet.write(row, 1, total)
    worksheet.write(row, 2, str(hapax/total))

    row += 2
    for result in results_list:
        worksheet.write(row, 0, result[0])
        worksheet.write(row, 1, result[1])
        row += 1
```

It then writes this into an xlsx file; no intermediate representation is used as currently the intermediate data is not expected to be needed in the immediate future.

3.10 syntactic_category_fetch.py

This module goes back into the dictionary to try to extract the syntactic category of each entry.

```
    import string
    import re
    import xlsxwriter
    import json

    INVALID_EXCEL_CHAR = r'[\[\]\:\*\?\/\\]'

    with open('dict.txt', encoding='utf-8') as f:

        res = dict()
        curr = ''

        for l in f:
            line = l.split(' ')

            try:
                if line[0].isupper():
                    curr = re.sub(r'\([^()]*\)', '', line[0])
```

```
                line = line[1:]

            for word in line:
                if word.islower() and '.' in word and not
                ↪  any([digit in word for digit in
                ↪  string.digits]):
                    word = re.sub(INVALID_EXCEL_CHAR, '',
                    ↪  word)
                    rlist = res.get(word, [])
                    rlist.append(curr)
                    res[word] = rlist
                else:
                    raise IndexError

        except IndexError:
            continue
```

This module takes much the same approach as `simple_get_dict_entry.py`. It looks for strings resembling abbreviations for parts of speech, which in the dictionary seemed to take the form of lowercase letters with at least one period. For each of these potential syntactic category markers, all entries found with it are kept track of. This method produces a lot of chaff, but has the advantage of being simple and fairly likely to be complete.

```
with open('../data/syncat.json', encoding='utf-8', mode='w')
↪  as f:
    json.dump(res, f)

workbook = xlsxwriter.Workbook('syntactic_category.xlsx')
for key in res:
    row = 0
    if len(res[key]) < 20:
        continue
    worksheet = workbook.add_worksheet(key)
    for entry in res[key]:
        worksheet.write(row, 0, entry)
        row += 1
workbook.close()
```

The data is then stored in a JSON file as well as written to an xlsx file.