Replicating the Paper: Learning the Structure of Biomedical Relationships from Unstructured Text

Authors: **Quan Nguyen**, Hui Wen Goh, Ruije Ma, Oumaima Bentakhou, Tanish Kumar, Nina Ghar

Thank you

We would like to thank Dr. Debaleena Das for this great opportunity. We have learned so much this summer, both in our technical skills and our soft skills.

Thank you Colin Magdamo for his guidance and patience. He offered many great technical advices in our projects and valuable viewpoints for industrial and academic applications.

Thank you Sarah Ryan-Purcell for organizing all the events and help us figure out the logictics.

Finally, we would like to thank Dr. Percha and Dr. Altman for their original research on this topic. The paper can be found here: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004216

Introduction

Currently, the process of discovering, developing, and approving a new drug takes about 12 to 15 years, with each drug costing ~$1 billion to reach the market. Our main goal is to reduce the time and cost of drug development. To achieve this goal, our team aims to discover latent drug-gene relationships in our existing biomedical texts.

In the early age of pharmacology, drugs were found through traditional medicines or by chance. After the early stage, classical (aka forward) pharmacology was used to investigate chemical libraries by testing them on animals and humans, then the drug-target of a working chemical was found later in the lab. In recent decades, an approach called reverse pharmacology allows for the discovery of promising target proteins before testing them on animals and humans. While there are more ways to discover drugs, as well as on-going debates on which approach is most efficient, we are mainly concerned with reverse pharmacology since it allows us to discover potential drug-gene relationships.

As of 2013, there were 23 million papers on the topic of phenotypes created by drug-gene interactions. These interactions are important for the understanding of drug responses and drug-drug interactions. While many knowledge database like OMIM, DrugBank, and PharmGKB are able to manually curate and restructure information to obtain various interactions, there exists no database that automatically obtain and use the vast amount of information available.

In our project, we aim to reduce the time and cost of discovering drug-gene relationships by using machine learning to comprehend existing biomedical texts.

Project Outline

We divided our project into 6 main tasks, with each task having their sub-sections.

1. Pull .xml files from PubMed baseline
2. Pre-processing all abstracts to obtain dependency matrix
   - Extract all abstracts from .xml files
   - Obtain the drugs list
   - Obtain the genes list
   - Filter abstracts for all sentences with matching drugs and genes pairs
   - Obtain the dependency matrix
   - Count and filter dependency matrix
3. Unsupervised Biclustering
   - Find the optimal k and l
   - Obtain the co-occurence matrix
4. Supervised Step
   - Obtain verified drug-gene pairs from DrugBank
   - Generate seed sets and test sets
   - Score test sets against seed sets
   - Assess the quality of unsupervised clustering
5. Scaling to Dask
   - Convert all .xml files to .csv files
   - Map all functions to Dask DataFrame
6. Creating Dendrogram with the Cluster Assignments
7. Final Notes

1. Pull .xml files from PubMed baseline

In order to process the .xml files, we need first need to acquire all .xml files from PubMed baseline database. We gathered the url for each .xml files, then requested the pull.

Since we are trying to replicate the original paper, we must use only abstracts from 2013. We figured out the range for the .xml files by using the way back machine to access the README.md in 2014, which stated that the final .xml for 2013 is "pubmed14n0777".

Link for PubMed baseline: https://ftp.ncbi.nlm.nih.gov/pubmed/baseline/

In [ ]:

```
import json # PubMed database searches retrieve results in JSON format. This module provides functions to acces
import urllib.request # The urlopen() function sends the request URL to the NCBI server to execute and retrieve
import time
import os
import requests
import pandas as pd
from bs4 import BeautifulSoup
from all_functions import *

url = 'https://ftp.ncbi.nlm.nih.gov/pubmed/baseline/'
reqs = requests.get(url)
soup = BeautifulSoup(reqs.text, 'html.parser')

urls = []
for link in soup.find_all('a'):
    #print(link.get('href'))
    urls.append(link.get('href'))

def download(url: str, dest_folder: str):
    if not os.path.exists(dest_folder):
        os.makedirs(dest_folder)  # create folder if it does not exist

    filename = url.split('/')[-1].replace(" ", "_")  # be careful with file names
    file_path = os.path.join(dest_folder, filename)

    r = requests.get(url, stream=True)
    if r.ok:
        print("saving to", os.path.abspath(file_path))
        with open(file_path, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024 * 8):
                if chunk:
                    f.write(chunk)
                    f.flush()
                    os.fsync(f.fileno())
    else:  # HTTP status code 4XX/5XX
        print("Download failed: status code {}\n{}".format(r.status_code, r.text))

pubmed_paths = []
print("here")
for link in urls[2:]:
    if not 'md5' in link:
        pubmed_paths.append("all_xml/"+link)
        download(url+link, dest_folder="all_xml/")
```

2. Pre-processing all abstracts to obtain dependency matrix

First, we must import all required packages. The usage of all packages will be explained in their appropriate sections.

In [ ]:

```
import os
import pandas as pd
import pubmed_parser as pp
import numpy as np
import re
import nltk
from nltk.tokenize import sent_tokenize as st
from nltk.corpus import stopwords
from nltk.parse import stanford
import random
from operator import itemgetter
from ebc import EBC
from matrix import SparseMatrix
import unittest
# from Bio import SeqIO
import spacy
```

Extract all abstracts from .xml files

To parse PubMed's .xml files, we used the "pubmed_parser" package. We extracted and created a pandas DataFrame containing only the abstracts of all the research papers.

In [ ]:

```
def get_all_abs(directory = 'all_xml/'):

    """
    Put all abstracts from .xml files into dataframe
    """

    # Get xml
    file_list = os.listdir(directory)
    # file_cur = file_list[0]

    # Parse medline xml to pandas
    file_df = None
    for i in range(len(file_list)):
        file_cur = file_list[i]
        if file_df is None:
            file_df = pd.DataFrame.from_dict(pp.parse_medline_xml(directory+file_cur))
        else:
            file_df = file_df.append(pd.DataFrame.from_dict(pp.parse_medline_xml(directory+file_cur)))

    # Filter out empty abstracts
    abs_df = file_df[['abstract']][file_df['abstract'] != '']
    abs_df = abs_df.apply(lambda x: x.astype(str).str.lower())

    return abs_df
```

Obtain the drugs list

Before we can parse the abstracts for drugs and genes pairs, we first need to create a list of drugs. We downloaded the "chemicals.tsv" file from DrugBank's database. We filtered the file to obtain only the verified drugs along with their main name, generic names, and trade names. The drug names are filtered again with a list of the most common English words to avoid obtaining meaningless relationships later in our pipeline.

The original paper generates a list of drugs with 4008 drug names, compared to ours with 26953 drug names. While the original paper doesn't show the exact filtering and name selection process, we think that the differences lie in the criteria for filtering the drugs. The original paper might have used a more rigorous filter, which resulted in a much shorter drug list.

To reduce noise in our drug list, we added an additional filter using a list of common biomedical words. For instance, we performed a dictionary count of all the words from all the abstracts, then picked out some of the most frequent words to filter against our current list. As of right now, we were able to generate the common biomedical words from 4 .xml files, but we have not done so for the entire .xml library due to time limitation.

In [ ]:

```
def get_common_words():

    """
    Obtain list of common words
    List include: common words from Education First, stop words from nltk, and the alphabet
    """

    with open('input/common_words_ef.txt','r') as f: # from 3000 common words
        freq_words = f.readlines()
    for i in range(len(freq_words)):
        freq_words[i] = freq_words[i].strip('\n')

    alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',

    freq_words = list(set(freq_words + stopwords.words('english')+alphabet))

    return freq_words

def get_drugs(freq_words):

    """
    Obtain drugs list from PharmGKB
    """

    drugs = pd.read_csv('input/chemicals.tsv',sep='\t')
    drugs = drugs[drugs['Type']=='Drug']

    drugs1 = pd.DataFrame(drugs['Name'], columns = ['Name'])
    drugs2 = pd.DataFrame(drugs['Generic Names'], columns=['Generic Names'])
    drugs3 = pd.DataFrame(drugs['Trade Names'], columns=['Trade Names'])
    drugs2 = drugs2.dropna()
    drugs3 = drugs3.dropna()

    for i in range(drugs2.shape[0]):
            drugs2_tmp = drugs2['Generic Names'].iloc[i].split(",")
            if len(drugs2_tmp) > 1:
                for j, item in enumerate(drugs2_tmp):
                    if j == 1:
                        drugs2['Generic Names'].iloc[i] = item
                    else:
                        drugs2 = drugs2.append({'Generic Names': item}, ignore_index = True)
    drugs2['Generic Names'] = drugs2['Generic Names'].apply(lambda x: re.sub("\s[\(].*?[\)]","",str(x).lower().

    for i in range(drugs3.shape[0]):
            drugs3_tmp = drugs3['Trade Names'].iloc[i].split(",")
            if len(drugs3_tmp) > 1:
                for j, item in enumerate(drugs3_tmp):
                    if j == 1:
                        drugs3['Trade Names'].iloc[i] = item
                    else:
                        drugs3 = drugs3.append({'Trade Names': item}, ignore_index = True)
    drugs3['Trade Names'] = drugs3['Trade Names'].apply(lambda x: re.sub("\s[\(].*?[\)]","",str(x).lower().stri

    drugs = pd.concat([drugs1, drugs2.rename(columns = {'Generic Names':'Name'}),
                       drugs3.rename(columns={'Trade Names':'Name'})]).reset_index(drop=True)

    for i in range(drugs.shape[0]):
            drugs_tmp = drugs['Name'].iloc[i].split(" / ")
            if len(drugs_tmp) == 1:
                drugs_tmp = drugs['Name'].iloc[i].split("/")
            if len(drugs_tmp) > 1:
                for j, item in enumerate(drugs_tmp):
                    if j == 1:
                        drugs['Name'].iloc[i] = item
                    else:
                        drugs = drugs.append({'Name': item}, ignore_index = True)
    drugs['Name'] = drugs['Name'].apply(lambda x: x.lower())
    drugs = [item for sublist in drugs.values.tolist() for item in sublist]
    drugs = [drug for drug in drugs if drug not in freq_words]
    drugs = [drug for drug in drugs if len(drug)>2]
    print(f'Length of drugs list = {len(drugs)}')

    return drugs
```

Obtain the genes list

Similar to the drugs list, we obtain the genes list from PharmGKB. The list contains the main names, the alternate names, and the symbols for the genes. The gene list is also filtered in the same manner as the drug list.

The original paper generated a gene list with 109597 terms, compared to ours with 130624 terms. The explaination for the difference is the same to that of the drugs list.

In [ ]:

```
def get_genes_pharmgkb(freq_words):

    """
    Obtain a list of genes
    This gene list from PharmGKB works better than the uniprot one
    Some genes are removed because:
        Similar to common words
        String length less than 2
    """

    genes = pd.read_csv('input/genes_pharmgkb.csv')
    genes1 = pd.DataFrame(genes['names+symbols'])
    genes2 = pd.DataFrame(genes['alternate_names+symbols'])
    genes2 = genes2.dropna()

    for i in range(genes2.shape[0]):
        genes2_tmp = genes2['alternate_names+symbols'].iloc[i].split(",")
        if len(genes2_tmp) > 1:
            for j, item in enumerate(genes2_tmp):
                if j == 1:
                    genes2['alternate_names+symbols'].iloc[i] = item.replace('"','')
                else:
                    genes2 = genes2.append({'alternate_names+symbols': item.replace('"','')}, ignore_index = Tr
    genes = pd.concat([genes1, genes2.rename(columns = {'alternate_names+symbols':'names+symbols'})]).reset_inc
    genes['names+symbols'] = genes['names+symbols'].apply(lambda x: str(x).lower())
    genes = [item for sublist in genes.values.tolist() for item in sublist]
    genes = [gene for gene in genes if gene not in freq_words]
    genes = [gene for gene in genes if len(gene)>2]
    print(f'Length of gene list = {len(genes)}')

    return genes
```

Filter abstracts for all sentences with matching drugs and genes pairs

The goal of this portion is to obtain all the sentences from the abstracts given lists of drugs and genes. Similar to the original paper, we filtered out sentences with length less than 4 or more than 50 words. For each abstract, we take all sentences with a drug-gene pair. This is done to account for all the variations that a drug and gene can occur in the same sentence.

We encounter some issues in extracting sentences. Some of them might not occur often, so we chose to skip these cases. Others require much more time to solve. Here, we are listing all the issues and potential solutions, if any.

1. Extracting sentences with non-relevant information (relevant = inhibit/increase/etc.)
   - We ignore this issue in hope that the next steps will get rid of these sentences
2. Cannot account for all variations of genes and drugs.
   - E.g. Endothelin-1 appears both as 'endothelin 1' and 'endothelin-1'.
3. Two genes occur in the same sentence.
   - E.g. In PBMCs from healthy volunteers, salbutamol did not modify IFN-gamma production, but increased the spontaneous production of IL-13.
4. Common biomedical words in drugs/genes lists interfered with filtering
   - Solution: Filtered out the most common biomedical words

In [ ]:

```
def get_filtered_data_all_sent(abs_df, drugs, genes):

    """
    Filter for abstracts + sentences containing drug+gene pairs
    input: dataframe containing only abstract, drugs, genes
    output: dataframe with 4 columns: abstract, sentence, drug, gene
    NOTE: this function takes all sentences with drug-gene pair in an abstract
    """

    abs_df = abs_df.apply(lambda x: x.astype(str).str.lower())

    abs_filt = pd.DataFrame(columns = ['abstract','sentence','drug','gene'])
    for index, abstract in abs_df.iterrows():
        try:
            for sentence in st(abstract.values[0]):
                if len(sentence.split()) < 4 or len(sentence.split()) > 50:
                    continue
                grabbed_sent = False
                tmp_sentence = sentence.replace(',','').replace('(','').replace(')','').replace('/',' ').replac
                for drug in drugs:
                    if ' '+drug+' ' in tmp_sentence or drug+' ' in tmp_sentence:
                        for gene in genes:
                            if ' '+gene+' ' in tmp_sentence or gene+' ' in tmp_sentence:
                                items = [abstract.values[0],sentence,drug,gene] # items to append to filtered c
                                abs_filt = abs_filt.append(pd.DataFrame([items],columns = ['abstract','sentence
                                grabbed_sent = True
                                break # break out of genes loop if a sentence is found
                    if grabbed_sent == True: # break out of drugs loop if a sentence is found
                        break
        except:
            continue
    return abs_filt
```

Obtain the dependency matrix

In this portion, the get_dependency_matrix takes in the list of sentences along with the corresponding drug-gene pairs. For each sentence and pair, the information is processed by the get_dependency_path code.

To obtain the dependency path, we must first implement the Stanford Parser. We used version 3.5.0 since this is the most likely used version in the original paper. Note that the original paper used Java, while we implemented Stanford Parser 3.5.0 through changing the Java pathing inside the nltk library. While most aspects of the parser remained the same, some notations are different, resulting in slightly different dependency paths.

Since we are obtaining paths from a large amount of data, there are many edge cases. While we were able to solve some of these cases, we do not know all the possibilities due to our time constraint. To process all sentences without having to check for the remaining edge cases, we use a try/except function to bypass the remaining errors. The result should still contain the majority of the sentences.

After obtaining the dependency paths from the sentences, we keep only those with length more than 2. We've found that some relation such as 'conj' and 'nsubj' provide no biomedical contexts.

Note that since the EBC code only takes matrices with a specific format, the output of our dependency matrix contains 3 columns: dependency paths, drug gene pairs, and relation (a column with all value as 1).

In [ ]:

```
def get_dependency_path(sentence, drug, gene):

    """
    Input: sentence, drug, gene
    Output: dependency path
    """

    java_path = r'C:\Program Files (x86)\Common Files\Oracle\Java\javapath\java.exe'
    os.environ['JAVAHOME'] = java_path
    os.environ['STANFORD_PARSER'] = r'C:\Users\nguye\Documents\Quan\stem-away\module 2 stanford parser\stanford
    os.environ['STANFORD_MODELS'] = r'C:\Users\nguye\Documents\Quan\stem-away\module 2 stanford parser\stanford
    dependency_parser = stanford.StanfordDependencyParser(path_to_jar=r'C:\Users\nguye\Documents\Quan\stem-away

    # cover edge case where [] and {} cannot be parsed
    sentence = sentence.replace('[','(').replace(']',')').replace('{','(').replace('}',')')

    try:
        result = dependency_parser.raw_parse(sentence)
        dep = next(result)

    except:
        print(f"Error parsing the following sentence:\n {sentence} \n------------------")
        result = []

    if result == []:
        return []

    # make dependency tuple into list
    try:
        dependency_list = []
        for relation in dep.triples():
            temp_list=[]
            for item in relation:
                if type(item).__name__ == 'tuple':
                    temp_list.append(str(item[0]))
                else:
                    temp_list.append(str(item))
            dependency_list.append(temp_list)
    except:
        return []

    # Obtain drug and gene path
    drug_path = []
    gene_path = []
    drug_path_search = []
    gene_path_search = []

    restart_loop = True
    loop_counter = 0
    while restart_loop == True:
        loop_counter += 1

        # specific cases where code keeps running while dependency_list is empty
        if len(dependency_list)==0 and loop_counter > 200:
            drug_path = []
            gene_path = []
            break

        for i in range(len(dependency_list)):
            relation = dependency_list[i]
            if relation[2] == drug:
                drug_path.append(relation[2])
                drug_path.append(relation[1])
                drug_path.append(relation[0])
                drug_path_search = relation[0]
                dependency_list.pop(i)
                loop_counter = 0
                break
            elif relation[2] == drug_path_search:
                drug_path.append(relation[1])
                drug_path.append(relation[0])
                drug_path_search = relation[0]
                dependency_list.pop(i)
                loop_counter = 0
                break
            elif relation[2] == gene:
                gene_path.append(relation[2])
                gene_path.append(relation[1])
                gene_path.append(relation[0])
                gene_path_search = relation[0]
                dependency_list.pop(i)
                loop_counter = 0
                break
            elif relation[2] == gene_path_search:
                gene_path.append(relation[1])
                gene_path.append(relation[0])
                gene_path_search = relation[0]
                dependency_list.pop(i)
                loop_counter = 0
                break
            elif i == (len(dependency_list)-1):
                restart_loop = False
                break

    # Combine drug and gene path into dependency path
    if drug in gene_path:
        ind = gene_path.index(drug)
        dependency_path = gene_path[1:ind]
    elif gene in drug_path:
        ind = drug_path.index(gene)
        dependency_path = drug_path[1:ind][::-1]
    else:
        dependency_path = gene_path[1:] + drug_path[::-1][1:-1]

    return dependency_path

def get_dependency_matrix(abs_filt):

    """
    Input: Dataframe of filtered abstract with columns of abstract, sentence, drug, and gene
    Output: Dependency matrix in the form: dependency_paths, drug_gene_pairs, relation (always 1)
    """

    dependency_matrix = pd.DataFrame(columns = ['dependency_paths','drug_gene_pairs','relation'])
    for i, sentence in enumerate(abs_filt["sentence"]):
        print('-------------------------')
        print(sentence)
        drug = abs_filt["drug"].iloc[i]
        gene = abs_filt["gene"].iloc[i]
        dependency_path = get_dependency_path(sentence, drug, gene)
#       print(drug)
#       print(gene)
#       print('----dependency path----')
#       print(dependency_path)
#       print('-----------------------')
        if len(str(dependency_path).split(',')) > 2 and dependency_path[0]!='conj':
            drug_gene_pair = drug+'/'+gene
            to_append = [dependency_path,drug_gene_pair,1]
            dependency_matrix = dependency_matrix.append(pd.DataFrame([to_append],columns = ['dependency_paths'
            print('append to matrix')
        print('done')

    return dependency_matrix
```

Count and filter dependency matrix

After acquiring all dependency paths, we want to filter our data so that sentences without drug-gene pairs relation are eliminated. This is achieve by filtering out all sentences or drug-gene pairs that occur less than 2 times (for sparse filtering) or less than 5 times (for dense filtering).

In [ ]:

```
def count_and_filter_dep_mat(dependency_matrix,num_to_filt=1):
    """
    Input: dependency_matrix, num_to_filt
        num_to_filt: how many times does a path or pair has to appear to be included in the matrix?
                     The paper suggests either "1" or "4"
    Output: dependency_matrix with reduced data
    """

    # Count pairs occurence and filter out those less than num_to_filt
    pairs_count_df = dependency_matrix['drug_gene_pairs'].value_counts().rename_axis('drug_gene_pairs').to_fram
    pairs_count_df = pairs_count_df[pairs_count_df.counts > num_to_filt]
    pairs_count_df['drug_gene_pairs'] = pairs_count_df.index
    pairs_for_filter = pairs_count_df['drug_gene_pairs'].tolist()

    dependency_matrix = dependency_matrix[dependency_matrix['drug_gene_pairs'].isin(pairs_for_filter)]

    # Count paths occurence and filter out those less than num_to_filt
    paths_count_df = dependency_matrix['dependency_paths'].value_counts().rename_axis('dependency_paths').to_fr
    paths_count_df = paths_count_df[paths_count_df.counts > num_to_filt]
    paths_count_df['dependency_paths'] = paths_count_df.index
    paths_for_filter = paths_count_df['dependency_paths'].tolist()

    dependency_matrix = dependency_matrix[dependency_matrix['dependency_paths'].isin(paths_for_filter)]
    dependency_matrix = dependency_matrix.reset_index(drop=True)

    return dependency_matrix
```

3. Unsupervised Biclustering

Find the optimal k and l

Before running the ITCC algorithm to obtain the cluster assignments for our drug-gene pairs (row) and dependency paths (column), we need to assign the number of row and column clusters, or k and l, respectively. Here we follow a heuristic created by the original paper.

Our goal is to minimize the objective function:

objective(M) - objective(M'),

where M is the original dependency matrix, while M' is the random dependency matrix. M' contains all rows and columns of M, but the locations of the rows and columns are randomized.

We find the optimal k and l as follow:

1. Create M' by randomizing the rows and columns locations of M.
2. Run both M and M' through EBC iteratively with a range of k and l.
3. The optimal k and l is the one with the lowest difference in the objective value from EBC (check objective equation).

In [1]:

```
import csv
from matrix import SparseMatrix
from ebc import EBC
import random
import pandas as pd

# Extract matrix for the original and shuffled dataset
def get_original_matrix(file):
    with open(file) as filename:
        reader = csv.reader(filename)
        data = list(reader)
        data = data[1:]
        data_size = len(data)
        matrix = SparseMatrix([data_size,data_size])
        matrix.read_data(data)
        matrix.normalize()
        return matrix


def get_random_matrix(file_r):
    with open(file_r) as filename:
        reader = csv.reader(filename)
        data = list(reader)
        data = data[1:]
        data_size = len(data)
        matrix_r = SparseMatrix([data_size,data_size])
        matrix_r.read_data(data)
        matrix_r.normalize()
        return matrix_r

def get_objective_value_k(k):
    ebc = EBC(matrix, [k, k], 10, 1e-10, 0.01)
    cXY, objective, it = ebc.run()
    return objective
def get_objective_value_k_r(k):
    ebc = EBC(matrix_r, [k, k], 10, 1e-10, 0.01)
    cXY, objective_r, it = ebc.run()
    return objective_r
def get_objective_value_l(l):
    ebc = EBC(matrix, [k_value, l], 10, 1e-10, 0.01)
    cXY, objective, it = ebc.run()
    return objective
def get_objective_value_l_r(l):
    ebc = EBC(matrix_r, [k_value, l], 10, 1e-10, 0.01)
    cXY, objective_r, it = ebc.run()
    return objective_r

def result_k(output_k,output_k_r,k_range):
    k_value = [i for i in k_range]
    df_result_k = pd.DataFrame(columns = ["k_value", "objective_value",'objective_value_r'])
    df_result_k["k_value"] = k_value
    df_result_k['objective_value'] = output_k
    df_result_k['objective_value_r'] = output_k_r
    df_result_k['difference'] = df_result_k['objective_value'] - df_result_k['objective_value_r']
    df_sorted_k = df_result_k.sort_values(by='difference')
    return df_sorted_k
def result_l(output_l,output_l_r,l_range):
    l_value = [i for i in l_range]
    df_result_l = pd.DataFrame(columns = ["l_value", "objective_value",'objective_value_r'])
    df_result_l["l_value"] = l_value
    df_result_l['objective_value'] = output_l
    df_result_l['objective_value_r'] = output_l_r
    df_result_l['difference'] = df_result_l['objective_value'] - df_result_l['objective_value_r']
    df_result_l.sort_values(by='difference')
    return df_result_l

def Main():
    matrix = get_original_matrix(file)
    matrix_r = get_random_matrix(file_r)
    output_k = []
    for k in k_range:
        output_k.append(get_objective_value_k(k))
    output_k_r = []
    for k in k_range:
        output_k_r.append(get_objective_value_k_r(k))
    df_k = result_k(output_k,output_k_r,k_range)
    k_value = df_k.iloc[0,0]
    output_l = []
    for l in l_range:
        output_l.append(get_objective_value_l(l))
    output_l_r = []
    for l in l_range:
        output_l_r.append(get_objective_value_l_r(l))
    df_l = result_l(output_l,output_l_r,l_range)
    l_value = df_l.iloc[0,0]
    return k_value,l_value

file = "dependency_matrix_dask.csv" # Input original dependency matrix here
file_r = "dependency_matrix_dask_shuffled.csv" # Input randomized dependency matrix here
matrix = get_original_matrix(file)
matrix_r = get_random_matrix(file_r)
k_range = (1,3,1)
l_range = (1,3,1)
k_value,l_value = Main()
```

Obtain the co-occurence matrix

The co-occurence matrix is an n by n matrix with all drug-gene pairs as the columns and rows. Each cell contains the number of occurence for one drug-gene pair to another after N times of running EBC. For our project, we set N=1000 since it is the optimal value according to the original paper. The co-occurence matrix simply counts all the occurence iteratively.

For the EBC portion, we used the package directly from Dr. Percha.

https://github.com/blpercha/ebc

In [ ]:

```
def get_co_occur_mat(dependency_matrix_count_filtered, n, optimal_k, optimal_l):

    """
    Input: dataframe of dependency matrix, n (EBC run n times), optimal_k, optimal_l
    Output: dataframe of co-occurence matrix
    """

    dep_mat_count_filtered = dependency_matrix_count_filtered.values.tolist()

    # set up EBC according to example EBC file
    mat_size = len(dep_mat_count_filtered)
    matrix = SparseMatrix([mat_size, mat_size])
    matrix.read_data(dep_mat_count_filtered)
    matrix.normalize()

    # run EBC n times
    rows_clusters = []
    for r in range(0,n):
        ebc = EBC(matrix, [optimal_k, optimal_l], 10, 1e-10, 0.01)
        cXY, objective, it = ebc.run()
        rows_clusters.append(cXY[0])

    co_occur_matrix = np.zeros((len(cXY[0]), len(cXY[0])))

    for c in rows_clusters:
        for i in range(0, len(c)):
            for j in range(0 , len(c)):
                if c[i] == c[j]:
                    co_occur_matrix[i][j] += 1

    pairs_list = [row[1] for row in dep_mat_count_filtered]
    co_occur_df = pd.DataFrame(data=co_occur_matrix, index=pairs_list, columns=pairs_list)

    return co_occur_df
```

4. Supervised Step

The supervised step assesses the quality of our unsupervised step clustering. The supervised step includes obtaining the verified drug-gene pairs, generating seed and test sets, and scoring the test sets against the seed sets.

Obtain verified drug-gene pairs from DrugBank

To obtain the drug-gene pairs from DrugBank, we must find a list of all the pharmacologically active relationships and a list of all the drugs. The file 'pharmacologically_active.csv' provides all the drug-gene pairs for humans, with all gene names and the drug IDs. While the 'drugbank_vocabulary.csv' provides the drug names corresponding to the drug IDs. Our code output a list of drug-gene pairs accouting for all variations in the genes and the drugs. This output list allows us to generate the seed and test sets.

In [ ]:

```
def get_drugbank_pairs(directory="input/"):
    """
    Returns a list of drug gene pairs obtained from drug bank
    Required files: pharmacologically_active.csv and drugbank_vocabulary.csv (both from DrugBank)
    """

    # read in data
    drug_gene_relation = pd.read_csv(directory+'pharmacologically_active.csv')
    drugs = pd.read_csv(directory+'drugbank_vocabulary.csv')

    # delete unnecessary columns
    drug_gene_relation = drug_gene_relation.drop(columns=['ID','Name','GenBank Protein ID','GenBank Gene ID','U
    drugs = drugs.drop(columns=['Accession Numbers','CAS','UNII','Standard InChI Key'])

    # Separate each Drug ID into individual row
    for i in range(drug_gene_relation.shape[0]):
            drugIDs_tmp = drug_gene_relation['Drug IDs'].iloc[i].split("; ")
            gene_tmp = drug_gene_relation['Gene Name'].iloc[i]
            if len(drugIDs_tmp) > 1:
                for j, drugID in enumerate(drugIDs_tmp):
                    if j == 1:
                        drug_gene_relation['Drug IDs'].iloc[i] = drugID
                    else:
                        drug_gene_relation = drug_gene_relation.append({'Gene Name': gene_tmp,'Drug IDs': drugI

    drugIDs = drugs['DrugBank ID'].tolist()
    common_name = drugs['Common name'].tolist()
    synonyms = drugs['Synonyms'].tolist()

    # Replace Drug IDs with their Common name drug_gene_relation.shape[0]
    drug_gene_relation["Synonyms"]=np.nan
    for i in range(drug_gene_relation.shape[0]):
        drugID_in_relation = drug_gene_relation['Drug IDs'].iloc[i]
        for j in range(len(drugIDs)):
            if drugID_in_relation == drugIDs[j]:
                drug_gene_relation['Drug IDs'].iloc[i] = common_name[j]
                drug_gene_relation['Synonyms'].iloc[i] = synonyms[j]
    drug_gene_relation = drug_gene_relation.rename(columns={"Gene Name": "genes", "Drug IDs": "drugs"})

    for i in range(drug_gene_relation.shape[0]):
        try:
            drugs_tmp = drug_gene_relation['Synonyms'].iloc[i].split(" | ")
            if len(drugs_tmp) > 1:
                for j, item in enumerate(drugs_tmp):
                    if j == 1:
                        drug_gene_relation['drugs'].iloc[i] = item
                    else:
                        drug_gene_relation = drug_gene_relation.append({'drugs': item, 'genes': drug_gene_relat
        except:
            continue

    drug_gene_relation = drug_gene_relation.drop(columns=['Synonyms'])

    drugbank_pairs = []
    for i in range(drug_gene_relation.shape[0]):
        drug = drug_gene_relation['drugs'].iloc[i]
        gene = drug_gene_relation['genes'].iloc[i]
        drug_gene_pair = str(drug).lower()+'/'+str(gene).lower()
        drugbank_pairs.append(drug_gene_pair)

    return drugbank_pairs
```

Generate seed sets and test sets

A seed set contains a list of verified drug-gene pairs, with the number of pairs varied from 1 to 100. A test set contains a list of 50% verified drug-gene pairs, and 50% extracted drug-gene pairs, with 100 pairs at all time. Note that between a pair of seed set and test set, there is no overlapping drug-gene pairs.

In [ ]:

```
def get_seed_set_and_test_set(drugbank_pairs, drug_gene_pairs, size):

    """
    Input: drug-gene pairs from drugbank, drug-gene pairs obtained from our code, size of seed_set
    Output: seed_set with input size and test_set with size=100
    """

    common_drug_gene = [value for value in drug_gene_pairs if value in drugbank_pairs]
    uncommon_drug_gene = list(set(drug_gene_pairs)-set(common_drug_gene))

    print(len(common_drug_gene)) # test how many common drug and gene we have wih drug bank

    # Obtain random seed sets and half of test sets from drugbank. Ensure no overlap
    ran_int = random.sample(range(len(drugbank_pairs)), size+50)
    seed_set = itemgetter(*ran_int[:size])(common_drug_gene)
    test_set_drugbank_half = itemgetter(*ran_int[size:])(common_drug_gene)

    # Obtain random half of test sets from filtered drug_gene_pairs
    ran_int = random.sample(range(len(uncommon_drug_gene)), 50)
    test_set_filtered_half = itemgetter(*ran_int)(uncommon_drug_gene)

    # Combine the two halves test sets
    test_set = test_set_drugbank_half + test_set_filtered_half

    return seed_set, test_set
```

Score test sets against seed sets

After generating a seed set and a test set, we rank the test set members based on their co-occurence with the seed set members. The final score Ti for a specific test set member is the rank sum of that member with all seed set members. The exact formula is found on page 22 in Dr. Percha's paper.

In [ ]:

```python
def ebc_score(seed_set, test_set, co_occur_mat):

    scores = {}

    for Ti in test_set:
        score_Ti = 0
        ranks = {}
        if Ti in co_occur_mat.columns.tolist():
            co_occur_mat_sorted_by_Ti = co_occur_mat.sort_values(by=[Ti], ascending=False)
            co_occur_mat_sorted_by_Ti = co_occur_mat_sorted_by_Ti.loc[co_occur_mat_sorted_by_Ti["drug_gene"] !=
            n = len(co_occur_mat_sorted_by_Ti['drug_gene'])
            j = 1
            for drug_gene in co_occur_mat_sorted_by_Ti['drug_gene']:
                ranks[drug_gene] = j
                j+=1
            for Rij in co_occur_mat_sorted_by_Ti['drug_gene']:
                if Rij in seed_set:
                    score_Ti += ranks[Rij]
        scores[Ti] = score_Ti

    return scores
```

Assess the quality of unsupervised clustering

The original paper generates randomly 1000 seed sets and test sets, resulting in 1000 different rankings. The rankings' quality is determined with the method below.

For each verified drug-gene pair ranking, we're supposed to determine if the ranking is true or false. Using this information, we can generate an ROC curve with the true-positive-rate on the y-axis and false-positive-rate on the x-axis. The AUC score, or the area under the ROC curve, provides us a measure of how likely it is that a positive element of the test set will be ranked higher than a negative element. The original paper considers any rankings with an AUC > 0.7 acceptable.

Notes:

- We are uncertained on how true or false is assigned to each ranking.
- The AUC value of 0.7 was chosen by the original author after observing the frequency of correct rankings based on their AUC value. The information can be found on page 125 in Dr. Percha's dissertation titled "Biomedical Text Mining from Context".

5. Scaling to Dask

In order to run all .xml files, we use Dask to scale up our operations. All core functions are kept the same. The main difference between our sequential computation and parallel computation is a conversion from .xml to .csv, and the mapping of the functions to the dask dataframe.

Convert all .xml files to .csv files

This step is necessary because Dask DataFrame requires a .csv input.

In [ ]:

```python
def convert_xml_to_csv(xml_file,directory):
    csv_file = pd.DataFrame.from_dict(pp.parse_medline_xml(xml_file))
    csv_file.to_csv(path_or_buf = directory+re.sub(r'.xml',r'.csv',xml_file[len(directory):-3]), index=False)
    pass

def convert_all_xml_to_csv(directory='all_xml/'):

    """
    Convert xml to csv
    Change the range to the length of all the files listed to get the full range
    directory: folder where xml are located
    """

    file_list = os.listdir('all_xml/')
    pubmed_paths = [directory+'pubmed21n'+str(num).zfill(4)+'.xml.gz' for num in range(1,len(file_list))]
    for xml_file in pubmed_paths:
        convert_xml_to_csv(xml_file,directory)
```

Map all functions to Dask DataFrame

The mapping is applied from the beginning up to the obtain dependency matrix step. After that, parallel computing is not necessary. However, with the many loops in the rest of the code, we could parallel all the steps after the dependency matrix with more time.

In [ ]:

```python
import dask
import dask.dataframe as dd
import spacy
import pandas as pd
import regex as re
import pubmed_parser as pp
import numpy as np
from all_functions import *
import os

#read in all csv as dask dataframe
df = dd.read_csv('all_xml/pubmed21*.csv', sample=25000000,
                 dtype={'title':'object','abstract':'object','journal':'object',
                        'authors':'object','pubdate':'object','pmid':'object',
                        'mesh_terms':'object','publication_types':'object','chemical_list':'object',
                        'keywords':'object','doi':'object','references':'object','delete':'object',
                       'affiliations':'object','pmc':'object','other_id':'object','medline_ta':'object',
                       'nlm_unique_id':'object','issn_linking':'object','country':'object'})

#take only the abstract column
abs_df = df[['abstract']]

#map filtering for all sentences containing drug-gene pair function
abs_filt = abs_df.map_partitions(lambda x: get_filtered_data_all_sent(x, drugs, genes))

#map obtaining matrix from sentences and drug-gene pairs
dependency_matrix = abs_filt.map_partitions(lambda x: get_dependency_matrix(x))

#compute all map function
dependency_matrix_pd = dependency_matrix.compute()
```

6. Creating Dendrogram with the Cluster Assignments

The dendrogram contains all the drug-gene pairs and how they are related. Imagine a family tree with each branch describing a relationship between family members. In our context, all the family members are drug-gene pairs and the relationships are within biomedical contexts (activator, inhibitor, etc.). With the cluster assignments, the "closeness" of the pairs are determined through the Spearman correlation.

After this step, we should be able to see which pairs have already been discovered, and which ones are completely new. The new drug-gene pairs are potential testing targets in the lab.

The code for this portion is in R.

7. Final Notes

Final bottleneck

While the entire pipeline works well, with both our sequential and parallel computing fully functional, we weren't able to process entirely 777 different xml files due to our limited hardware. Our personal computers ran out of memory after a few days of processing. Our project would be finished completely with either better hardware or more time to clean up the data structure of the code.

Future improvements

Our project focused mainly on replicating Dr. Percha's paper. However, we have some ideas to improve the current process.

1. Use the new Stanford Parser in the form of the stanza package.
2. With the stanza package, we can use the AWS server to compute all the xml files, eliminating the final bottleneck.
3. Use all updated xml files.