# Drug Recommender System
using Knowledge Graph Embedding

**Ananya Ramesh**
120220919

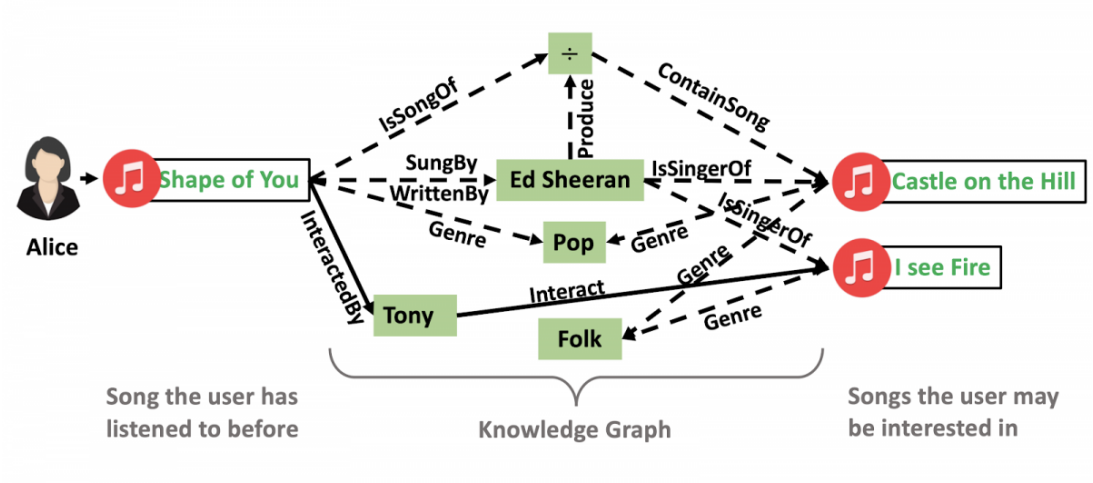
A report presented for
AM4065 - Network Theory

Applied Mathematics
University College Cork
Ireland
12 December 2021

# Contents

# 1 Introduction

In today's world, with technology assisting us in every professional field fathomable, a recommender system has become an integral part of our day-to-day activities. A recommendation system can be defined as a part of information filtering system which can help in predicting the "rating" or "preference" a user would give to an item. We encounter its application, with common examples like the form of playlist recommendations by various apps like YouTube, Spotify based on the songs we previously listened to and product recommenders for online stores like Amazon which compares the profile of an user to similar to what we have purchased. The Figure 1 illustrates how a knowledge graph looks for playlist generation.




From Research, Meteorology to Entertainment, Recommendation Systems have helped optimise the experience and processes based on the parameters required, thereby streamlining and continually learning at the same time. One industry which will benefit the most from such a recommendation system is the Healthcare and Medicine Industry. Since the advent of the internet, common people have tried to find cures for their ailments by just scouring the web which is harmful and not at all recommended by health professionals. The results of a survey by Pew Internet and American Life Project in 2013 [1] showed that 59% of adults have looked online for health topics, and with 35% of respondents focusing on diagnosing a medical condition online. But what if a system which has an extensive knowledge of medicines mapping them to disease, adverse drug-interactions helped medical professionals in recommending medicines to their patients by matching the patient profile to the medicine required. This may seem like a redundant process, but various studies have thrown a light on why Healthcare Industry sorely need such a system. The motivation of this project is to build a system to recommend medicines to a patient with a set of diagnoses. This project is mainly based on the research paper SMR: Medical Knowledge Graph Embedding for Safe Medicine Recommendation[2] and tries to implement the idea explained in the paper.

## 1.1 Industry Need for a patient-oriented drug recommendation system

More than 200 thousand people in China and 100 thousand people in USA die each year due to medication errors as per the respective administration's reports. According to a report published by HIQA[3] in 2018, an estimated 3 million medication errors happen in Irish hospitals each year. The extent of the issue is massive, and the cause of the problem can be attributed to these 3 reasons:

1. **Lack of experts for Critical Illness:** Chronic and Critical Illnesses are found everyday with varying severity, diagnosis and patient handling processes. It is difficult for hospitals to have experts in all these illnesses to provide optimal healthcare solutions to the patients.
2. **Lack of Resources other than experience:** More than 42% medication errors are caused by doctors because experts write the prescription according to their experiences. Though their experience might be an asset, it is not infallible due to the sheer number of variations in diseases and diagnosis.

3. **Lack of Robust Patient-Medicine compatibility verification:** Medicines come with a long list of side-effects and are even lethal to patients with specific allergies. With little to no system checking the compatibility of Patient History to Medicine, patients are affected even without medication errors.

Due to these endemic problems in the industry, a universal patient-oriented Recommendation System is not just a guiding tool but a necessity today, especially in the pandemic.

# 2 Terminologies

## 2.1 Knowledge Graph

A knowledge graph is a knowledge base which is implemented using a graph-structured data model to link data and it's properties. Knowledge graphs stores interlinked descriptions of entities – objects, events, situations or abstract concepts – while also encoding the semantics underlying the used terminology. They are also prominently associated with and used by search engines such as Google, Bing, and Yahoo; knowledge-engines and question-answering services such as WolframAlpha, Apple's Siri, and Amazon Alexa; and social networks such as LinkedIn and Facebook.[4]

In Knowledge Graph, we use different terminologies compared to the traditional vertices and edges used in graphs. The vertices of a knowledge graph are called entities and the directed edges are called triplets. They are represented as a *(h, r, t)* tuple, where h is the head entity, t is the tail entity, and r represents the relation between the head with the tail entities. In a homogeneous graph, all the entities belong the same category and the relations represent a single type. On contrast, a heterogeneous graph is where both the entities and relations belong to several different categories.

In this paper we use a heterogeneous graph to deduct drug-drug interactions between drugs prescribed to a patient for two or more different diagnoses. In this, heads and tails belong to the same category medicines whereas the relations belong to different types based on the interaction. We also use a homogeneous knowledge graph to link different diseases with each other which is used to calculate embedding for a new patient.

## 2.2 Knowledge Graph Embedding

To work with Knowledge Graphs, we need to understand the concept of embedding which a mathematical object/structure to cast the set of nodes h_1, h_2, ..., h_k into a k-dimensional vector h. KG embedding mainly consists of three steps:

1. Initial representation of entities (heads and tails) and relations

   This step specifies the form in which entities and relations are represented in a continuous vector space. Entities can be shown as vectors, i.e., deterministic points in the vector space. Relations are typically taken as operations in the vector space, which can be represented as vectors, matrices, tensors, multivariate Gaussian distributions, or even mixtures of Gaussian.

   To begin with, the algorithm first assign random values to embedding vectors of the entities and relations

2. Define scoring function

   A scoring function $f_r(h,t)$ is defined on each fact (h; r; t) to measure its plausibility. Ideas observed in the KG tend to have higher scores compared to those that have not been observed.

   The algorithm then uses the defined scoring function and picks a training set to continuously optimise the embeddings and stops only when a certain condition is met. For each iteration, a sample batch is taken out of the training set which contains a random corrupted fact for each triple. A random corrupted fact is a triple formed by substituting the head or the tail or both with another entity that makes the fact false. This method is known as negative sampling.

3. Learning relation and entity representation

   At last, to learn those entity and relation representations (i.e., embeddings), the final step solves an optimization problem that maximizes the total plausibility of observed facts

The scoring function is further optimized by adding the triple and the corrupted triple to the training batch thereby updating the embeddings.

Thus, semantic meaning is extracted from the knowledge graph at the end of the algorithm.

The embedding techniques can be broadly classified into: translational distance models and semantic matching models. In this paper, we use translational models specifically TransR and TransD models. To give an overview of translational model, it was inspired by the idea of translation invariance introduced in word2vec. A pure translational model is based on the fact that after applying a relational translation, the embedding vector of the entities are close to each other in the geometric space in which they are defined. To put in simple words, given a fact, the sum of embedding of head and the embedding of relation, the expected result would be the embedding of the tail. The closeness of the entities embedding is given by some distance measure and quantifies the reliability of a fact.[5]

### 2.2.1 TransR

In TransR[6], entity vectors and relation vectors are separated into two different spaces namely $R_d$, $R_k$ respectively. Then a relational projection matrix is used to translate the entity embedding into relational embedding. Note that, the dimensions of entity embeddings and relation embeddings are not necessarily identical, i.e., $k \neq d$

Given a triplet (h; r; t), TransR projects the entities h and t into the space specific to relation r, i.e., for each relation r, we set a projection matrix $\mathbf{M_r} \epsilon \mathbb{R}^{\mathbf{k}\times\mathbf{d}}$ , that projects entities from entity space to relation space.

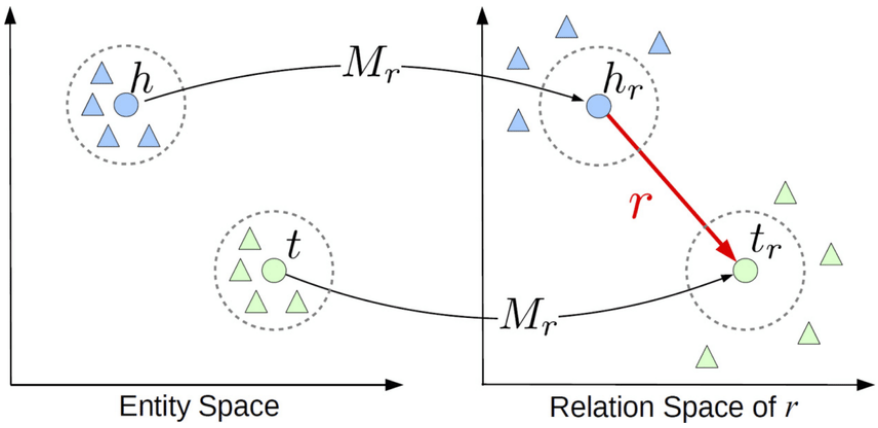
With this mapping matrix, we define the projected vectors of entities as

$$h_r = h \cdot M_r \qquad t_r = t \cdot M_r$$

The score function is correspondingly defined as

$$f_r(h,t) = \|h_r + r - t_r\|_2^2$$

The Figure 2 picture gives us a clear understanding of TransR which illustrates that for each triple (h, r, t), entities from the entity space are projected into r-relation space as $h_r$ and $t_r$ with operation $M_r$, and then $h_r + r \approx t_r$




### 2.2.2 TransD

TransD[7] simplifies TransR by decomposing the projection matrix into a product of two vectors. The first one is used to capture the meaning of entity and relations, the other one is used to construct mapping matrices.

For given triplet (h, r, t), we set two mapping matrices $M_{rh}, M_{rt} \epsilon \mathbb{R}^{\mathbf{m}\times\mathbf{n}}$ to project entities from entity space to relation space.

They are defined as follows:

$$M_{rh} = r_p \cdot h_p^T + I^{m \times n}$$

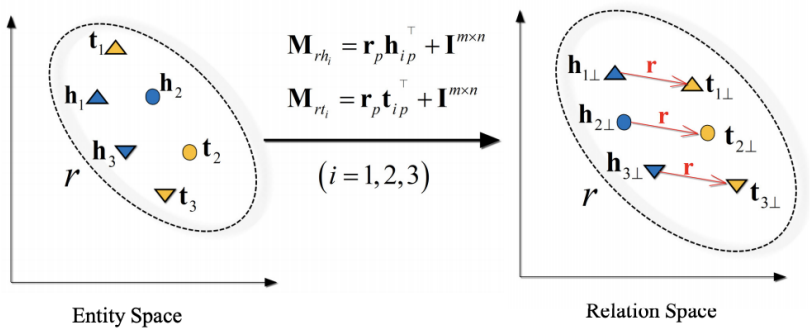
$$M_{rt} = r_p \cdot t_p^T + I^{m \times n}$$

where subscript $p$ marks the projection vectors

The above two projection matrices are applied on the head entity h and the tail entity t respectively to get their projections

$$f_r(h,t) = -\|M_{rh}h + r - M_{rt}t\|_2^2$$

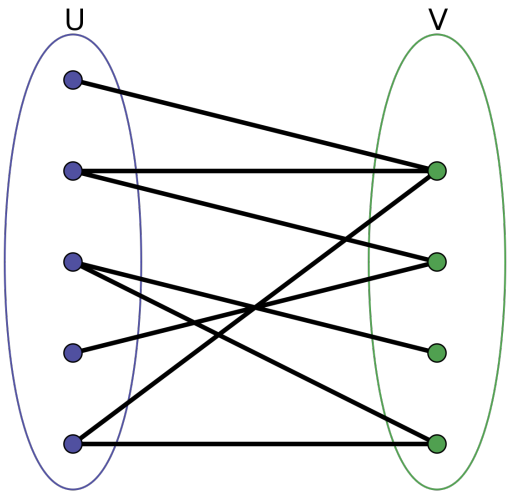
The Figure 3 illustrates how TransD considers different types of both entities and relations, to encode knowledge graphs into embedding vectors via dynamic mapping matrices produced by projection vectors.




## 2.3 Bipartite Graph

A bipartite graph is a graph where vertices are be divided into two disjoint and independent sets U and V such that every edge in U connects a vertex in V. Vertex sets U and V are usually called the parts of the graph. A bipartite graph is a special case of a k-partite graph with k=2. A bipartite graph also does not contain any odd-length cycles.

The Figure 4 shows two sets U denoted by blue color and V denoted by green color.




A bipartite graph is usually denoted by G=(U,V,E) where U and V represents the two sets and with E denoting the edges of the graph. Each edge $e \epsilon E$ is an ordered pair e = (u, v) and has a weight $w_{uv} >$

0, which indicates the strength of the relation. If a graph is undirected, we have $(u, v) \equiv (v, u)$ and $w_{uv} \equiv w_{vu}$ and if a graph is directed, we have $(u, v) \neq (v, u)$ and $w_{uv} \neq w_{vu}$. If a bipartite graph is not connected, it may have more than one bipartition

## 2.4 LINE Embedding

Large-scale Information Network Embedding(LINE)[8] is one of the embedding models that gives state-of-the-art results for a bipartite graph. To understand how LINE is implemented we first go through first-order and second-order proximity

Definitions for first-order and second-order proximity provided by [8]

> **First-Order Proximity** The first-order proximity in a network is the local pairwise proximity between two vertices. For each pair of vertices linked by an edge (u, v), the weight on that edge, $w_{uv}$, indicates the first-order proximity between u and v. If no edge is observed between u and v, their first-order proximity is 0.
>
> **Second-Order Proximity** The second-order proximity between a pair of vertices (u, v) in a network is the similarity between their neighborhood network structures. Let $p_u = (w_u, 1, ..., w_u, |V|)$ denote the first-order proximity of u with all the other vertices, then the second-order proximity between u and v is determined by the similarity between $p_u$ and $p_v$. If no vertex is linked from/to both u and v, the second-order proximity between u and v is 0.

### 2.4.1 LINE using Second-Order Proximity

The second-order proximity can be implemented for both directed and undirected graphs. In our implmentation, we construct an undirected bipartite graph. Since the second-order proximity works on the similarity if any two vertices share a certain number of connections to other vertices then the two vetices are concluded as similar to each other. Therefore, every vertex plays two roles

- The vertex being itself
- It plays a role of being a "context" to other vertices

We introduce two vectors $m_j$ and $\hat{m}_j$, where $m_j$ stands for medicine vector of j-th medicine and $\hat{m}_j$ stands when j-th medicine is used as a context. We also denote a patient vector of i-th patient as $p_i$. Then, the conditional probability is given by

$$P(mj|pi) = \frac{exp\{z(p_i, m_j)\}}{\sum_{\hat{m}_j \epsilon P_i} exp\{z(p_i, \hat{m}_j)\}}$$

# 3 Challenges of Medicine Recommender System using TransR embedding

With the SMR[2] model, there are few challenges that the system faces today:

1. Efficiency The computational time taken to calculate the embedding vectors of each triplet is very high and thus with the increasing number of medicines and drug-drug interactions, the complexity of the heterogeneous graph increases and becomes impossible to calculate the embeddings.
2. Data incompleteness The data taken from Drugbank[9] to construct the knowledge graph is not extensive and due to this we cannot perform a comprehensive interaction check between various medicines.
3. Cold Start The system works with already prescribed records and its importance. With the increasing rate of developing new medicines, the system cannot keep up with the frequent updates.

# 4 Datasets

We primarily use three datasets from three different sources to construct our graph for the recommendation system:

1. DrugBank [9] which is a xml file containing the list of all medicines with synonym, drug-drug interaction
2. MIMIC III [10] gives a comprehensive list of csv files containing various diagnoses, prescribed medicines, ICU admissions, labaratory tests, hospital details etc.. We use 2 files of these namely
   - PRESCRIPTIONS.csv - file containing the list of patients and the name of drug prescribed to them
   - DIAGNOSES_ICD.csv - file containing the list of patients and the ICD9 code of the disease they are diagnosed with.
3. IC9 Ontology [11] gives a csv file namely ICD9CM.csv which contains an extensive list of diseases along with the parent disease it belongs to.

# 5 Graph Construction Process

1. Construct a knowledge graph for predicting drug-drug interaction using the dataset from DrugBank[9]. The dataset will be of the format

```
<name>Lepidurin</name>
<synonyms>
    <synonym language="" coder="">2',3'-Dideoxycytidine</synonym>
    <synonym language="" coder="">
    4-amino-1-[(2R,5S)-5-(Hydroxymethyl)tetrahydrofuran-2-yl]pyrimidin-2H)-one
    </synonym>
    <synonym language="" coder="">DDC</synonym>
    <synonym language="" coder="">DDCYD</synonym>
    <synonym language="" coder="">Dideoxycytidine</synonym>
</synonyms>
<drug-interactions>
    <drug-interaction>
    <drugbank-id>DB00001</drugbank-id>
    <name>Apixaban</name>
    <description>
    Lepidurin increases the anticoagulant activites when combined with Apixaban.
    </description>
    </drug-interaction>
</drug-interactions>
```

   From this XML, we use the <name> tag as head nodes with <name> tag inside the <druginteraction> is used as tail node. The relationship is defined by the description after removing the drug name, interacted drug name and stop words. With this, we arrive at triples (h,r,t) which is used in constructing the graph.

2. Construct a knowledge graph for learning disease embedding from IC9 dataset[11] The dataset contains the ICD9 drug code with url and special character like ẇhich are removed for this implementation and the corresponding drug code of its parent. With this, we arrive at triple (di , 'parentOf', dj) where both the nodes h and t are diseases here.
3. Construct a bi-partite graph for patient-medicine using the dataset from Prescriptions.csv. Since the medicine names are in string format, it can be noisy due to the presence of words like 0.9% ml, mL etc... For this implementation, we use the synonym from the Drugbank to check for the medicine names and replace it with the actual drug name in the XML file.
4. Construct a bi-partite graph for patient-disease from the file Diagnoses_ICD.csv. This file consists of the patient id and icd9 code of the disease. Thus, it can be linked directly to the disease knowledge graph.

We use the above explained embedding technique to calculate TransD embedding of the two knowledge graphs and LINE embedding for the two bipartite graph. We finally add the embeddings to get 4 embeddings:

1. medicine_embedding - Embedding of each medicine combined with embedding from medicine knowledge graph and the weightage of number of times prescribed to patients
2. patient_embedding - Embedding of each patient combined from patient-disease bipartite graph and patient-medicine bipartite graph
3. disease_embedding - Embedding of each disease combined with embedding from disease knowledge graph and weightage of diagnosed for a patient
4. med_rel_embedding - Embedding of each unique relation found between drugs

We use the above calculated embeddings to recommend drugs to a given patient with list of diseases.

Consider a query = $[(p, d_1), (p, d_2)]$ where p is the patient id and $d_1 and d_2$ are the diseases that the patient is diagnosed with. First, we get the useful drugs for the respective diseases, and then calculate the score of each medicine prescribed to the patient given by

$$s[m_n] = p^T \cdot m_n$$

where p is the patient embedding and $m_n$ is the embedding vector of the n-th medicine

Then, the drug-drug interaction is calculated for each medicine from the first disease medicines(group 1) and second disease medicines(group 2)

$$ddi[m_n] = \sum_{j=1}^{k} \|m_i + r - m_j\|_{L1}, n \epsilon G_1, j \epsilon G_2$$

where k is the length of Group 2 medicines and L1 denotes L1 norm of the vector calculated.

The final score of the medicines are calculated as

$$score(m_n) = s[m_n] - ddi[m_n]$$

# 6 TransD comparison to TransR

The SMR[2] paper invloves TransR embedding to calculate the embedding of drug-drug interaction. Although TransR is powerful in modeling complex relations, it involves a projection matrix for each relation, that requires O(dk) parameters per relation where d is the number of entities and k is the number of relations. The total number of parameters required by TransR would be O(nd + mdk). So it loses the simplicity and efficiency of TransE/TransH which model relations as vectors and require only O(d) parameters per relation. But TransE and TransH uses simple vector sum to calculate the embeddings due to which they do not perform well with different types of knowledge graphs such as one-to-many, many-to-one and assymmetric ones. Thus, we cannot implement TransE/TransH even though it has simpler calculations.

But when we use TransD which breaks down TransR much more effectively, as the single projection matrix is broken down into two projection vectors, the total number of parameters required would be O(nd + nk).

We could also see the time taken by the package pykg2vec [12] to calculate embeddings for the same medicine dataset is tabulated as

| Method | Time Taken |
|---|---|
| TransR | 12 hours |
| TransD | 5.5 hours |

From this we can conclude TransD is an improved model of TransR. TransR directly defines a mapping matrix for each relation, TransD constructs two mapping matrices dynamically for every triplet by creating a projection vector for each entity and relation. In addition, TransD eliminates the matrix-vector multiplication operation and replaces it by vector operations. Without loss of generality, we assume $m \geq n$, the projected vectors can be computed as follows:

$$h_{\perp} = M_{rh} \cdot h = h_p^T \cdot h \cdot r_p + [h^T, 0^T]^T$$

$$t_{\perp} = M_{rt} \cdot t = t_p^T \cdot t \cdot r_p + [t^T, 0^T]^T$$

Therefore, TransD has less calculation than TransR, which makes it train faster and can be easily applied on large-scale knowledge graphs. This helps in reducing the time taken and complexity by half by shifting to TransD model. This model helps in tackling one of the challenge - Computational Efficiency the SMR model faces today. The newly proposed model can be scaled in a much easier way compared to SMR.

# 7 New Patient Model

The SMR model also proposes a way to calcuate the new patient embedding if a given list of diseases diagnosed for the patient are given in the order. It is calculated using the formula

$$p = \sum_{t=1}^{n} exp(-t) \cdot d_t$$

where $d_t$ is the disease embedding of the t-th disease.

# 8 Implementation

Implementation of the above mentioned model is done in python. The two packages used to calculate the TransD embedding and LINE embedding are pykg2vec[12] and graph-embedding repo [13] respectively. A few modifications have been made to the LINE embedding repo since it had outdated Tensorflow code and was changed to suit the old version.

The hyperparameters used for TransD embedding are :-

```
lmbda:0.1 #bias
batch_size:128
optimizer:adam #implements stochastic gradient descent
sampling:uniform
neg_rate:1 #to implement negative sampling
epochs:100
learning_rate:0.01 #learning rate for sgd
ent_hidden_size:50 #number of dimensions to have in the dimension vector
```

The parameters used for LINE embedding are:-

```
proximity='second-order' #as mentioned above, using second-order proximity
embedding_dim=50 #number of dimensions
num_batches=100 #same as epochs
batch_size=5
learning_rate=0.001 #learning rate for optimizer
```

[299]:
```
import numpy as np
import pandas as pd
import networkx as nx
from networkx.algorithms import bipartite
import xml.etree.ElementTree as ET
import tensorflow as tf
from argparse import Namespace
from sklearn.model_selection import train_test_split
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import math
nltk.download('stopwords')
nltk.download('punkt')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\anany\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\anany\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

[299]: True

[307]:
```
# This function is to remove stopwords to create relation in KG
def remove_stopwords(text):
    stop_words = set(stopwords.words('english'))
    word_tokens = word_tokenize(text)
    filtered_sentence = [w for w in word_tokens if not w.lower() in stop_words]
    return filtered_sentence
```

[116]:
```
#Parse XML
tree = ET.parse('medicine_kg.xml')
root = tree.getroot()
medicine_kg = []
synonym_ref = {}
for i in range(len(root)):
    drug_name = root[i].find("{http://www.drugbank.ca}name").text

    synonym_ref[drug_name] = []
    # This is needed for comparing the names in prescription.csv with drugbank id to␣
↪form patient medicine bipartite graph
    synonym_element = root[i].find("{http://www.drugbank.ca}synonyms")
    for synonym in synonym_element:
        if synonym.text is not None:
            synonym_ref[drug_name].append(synonym.text.lower())
    if len(synonym_ref[drug_name]) == 0:
        synonym_ref[drug_name].append(drug_name.lower())

    # This is needed for finding drug to drug interaction and constructing the␣
↪medicine knowledge graph
    interaction_element = root[i].find("{http://www.drugbank.ca}drug-interactions")


    for interaction in interaction_element:
        interaction_name = interaction[1].text
        description = interaction[2].text
        replace_arr = ["may", "combined", ".","risk", "severity", ",", 'used',␣
↪'combination'] + drug_name.lower().split(" ") + interaction_name.lower().split("␣
↪")

        description_arr = remove_stopwords(description)
        cleaned_description = [w for w in description_arr if not w.lower() in␣
↪replace_arr]
        find_adj = ["increases", "increased", "increase", "decreases", "decreased",␣
↪"decrease"]
        found_adj = [x for x in cleaned_description if x in find_adj]

        if len(found_adj) > 0:
            cleaned_description.remove(found_adj[0])
            if found_adj[0].find('increase') != -1:
                cleaned_description.insert(0, 'increases')
            else:
                cleaned_description.insert(0, 'decreases')
        relation = {
            'source': drug_name,
            'edge': ' '.join(cleaned_description),
            'target': interaction_name

        }
        medicine_kg.append(relation)
```

[143]:
```
# Splitting medicine knowledge graph triplets to train, test and valid to calculate␣
↪embedding by TransD
medicine_kg_df = pd.DataFrame.from_records(medicine_kg)
X_train, X_test_val = train_test_split(medicine_kg_df, test_size=0.6)
X_test, X_val = train_test_split(X_test_val, test_size=0.5)
X_train.to_csv('./med-train.csv',sep='\t')
X_val.to_csv('./med-valid.csv',sep='\t')
X_test.to_csv('./med-test.csv',sep='\t')
```

[144]:
```
# MIMIC iii prescription has patient to drug
prescription_df = pd.read_csv('PRESCRIPTIONS.csv')
prescription_df = prescription_df[['subject_id', 'drug', 'dose_val_rx',␣
↪'dose_unit_rx', 'form_unit_disp']]
prescription_df = prescription_df[(prescription_df['form_unit_disp'] != 'SYR') &␣
↪(prescription_df['form_unit_disp'] != 'BAG') & (prescription_df['form_unit_disp']␣
↪!= 'CAN')]
prescription_df = prescription_df[prescription_df['drug'].notna()]
prescription_df['drug'] = prescription_df['drug'].str.lower()
```

[145]:
```
# Since the drugs are not in standard form, look in synonym from drugbank
drugs = prescription_df['drug'].values
drugs = list(set(drugs))
final_drugs = {}
synonyms = list(synonym_ref.values())
actual_drug = list(synonym_ref.keys())
for i in drugs:
    for j in range(0, len(synonyms)):
        if i in synonyms[j]:
            final_drugs[i] = actual_drug[j]
```

[146]:
```
col1 = list(final_drugs.keys())
col2 = list(final_drugs.values())
data = {'drug': col1, 'actual_drug': col2}
drug_df = pd.DataFrame.from_dict(data)
drug_df = drug_df[drug_df['drug'].notna()]
```

[147]:
```
# Replace the drug name with the drugname in drugbank to connect to medicine␣
↪knowledge graph
final_df = pd.merge(prescription_df, drug_df, how='left', left_on='drug',␣
↪right_on='drug')
final_df = final_df[final_df['actual_drug'].notna()]
final_df.drop('drug', 1, inplace=True)
final_df.rename(columns={'actual_drug': 'drug'}, inplace=True)
```

[148]:
```
# Create count column with number of times the drug prescribed to the patient
pm_df = final_df.groupby(['subject_id', 'drug']).size().to_frame('count').
↪reset_index()
```

[149]:
```
# Construct bipartite graph
G = nx.Graph()
G.add_nodes_from(pm_df['subject_id'], bipartite=0)
G.add_nodes_from(pm_df['drug'], bipartite=1)
G.add_weighted_edges_from(
    [(row['subject_id'], row['drug'], row['count']) for idx, row in pm_df.
↪iterrows()], weight='weight')
```

[150]:
```
class DataLoader:
    def __init__(self,G):
        self.g=G
        self.num_of_nodes=self.g.number_of_nodes()
        self.num_of_edges=self.g.number_of_edges()
        self.edges_raw=list(self.g.edges(data=True))
        self.nodes_raw=list(self.g.nodes())
        self.edge_distribution=np.array([attr['weight'] for _,_,attr in self.
↪edges_raw],dtype=np.float32)
        self.edge_distribution/=np.sum(self.edge_distribution)
        self.node_negative_distribution=np.power(np.array([self.g.degree(node) for␣
↪node in self.nodes_raw],dtype=np.float32),0.75)
        self.node_negative_distribution/=np.sum(self.node_negative_distribution)
        self.node2idx={}
        self.idx2node={}
        for idx,node in enumerate(self.nodes_raw):
            self.node2idx[node]=idx
            self.idx2node[idx]=node
        self.edges=[(self.node2idx[u],self.node2idx[v]) for u,v,_ in self.edges_raw]

    def fetch_batch(self,batch_size=5,K=3):
        edge_batch_idx=np.random.choice(self.num_of_edges,size=batch_size,p=self.
↪edge_distribution)
        u_i=[]
        u_j=[]
        label=[]
        for edge_idx in edge_batch_idx:
            edge=self.edges[edge_idx]
            if np.random.rand()>0.5:
                edge=(edge[1],edge[0])
            u_i.append(edge[0])
            u_j.append(edge[1])
            label.append(1)
            for i in range(K):
                while True:
                    negative_node=np.random.choice(self.num_of_nodes,p=self.
↪node_negative_distribution)
                    if not self.g.has_edge(self.idx2node[edge[0]],self.
↪idx2node[negative_node]):
                        break
                u_i.append(edge[0])
                u_j.append(negative_node)
                label.append(-1)
        return u_i,u_j,label

    def embedding_mapping(self,embedding):
        return {node:embedding[self.node2idx[node]] for node in self.nodes_raw}
```

[151]:
```
class LINE:
    def __init__(self,args):
        tf.compat.v1.disable_eager_execution()
        self.u_i=tf.compat.v1.placeholder(dtype=tf.int32,shape=[args.
↪batch_size*(args.K+1)],name='u_i')
        self.u_j=tf.compat.v1.placeholder(dtype=tf.int32,shape=[args.
↪batch_size*(args.K+1)],name='u_j')
        self.label=tf.compat.v1.placeholder(dtype=tf.float32,shape=[args.
↪batch_size*(args.K+1)],name='label')
        with tf.compat.v1.variable_scope('embedding',reuse=True):
            self.embedding=tf.Variable(tf.compat.v1.truncated_normal([args.
↪num_of_nodes,args.embedding_dim]))
        self.u_i_embedding=tf.nn.embedding_lookup(self.embedding,self.u_i)
        if args.proximity=='first-order':
            self.u_j_embedding=tf.nn.embedding_lookup(self.embedding,self.u_j)
        elif args.proximity=='second-order':
            with tf.compat.v1.variable_scope('context_embedding',reuse=True):
                self.context_embedding=tf.Variable(tf.compat.v1.
↪truncated_normal([args.num_of_nodes,args.embedding_dim]))
            self.u_j_embedding=tf.nn.embedding_lookup(self.context_embedding,self.
↪u_j)
        self.inner_product=tf.reduce_sum(self.u_i_embedding*self.
↪u_j_embedding,axis=1)
        self.loss=-tf.reduce_mean(tf.compat.v1.log_sigmoid(self.label*self.
↪inner_product))
        self.learning_rate=tf.compat.v1.placeholder(dtype=tf.
↪float32,name='learning_rate')
        self.optimizer=tf.compat.v1.train.RMSPropOptimizer(learning_rate=self.
↪learning_rate)
        self.train_op=self.optimizer.minimize(self.loss)
```

[152]:
```
# Calculating LINE embedding for the bipartite graph
def train(graph):
    args=Namespace()
    args.proximity='second-order'
    args.embedding_dim=50
    args.num_batches=100
    args.K=3
    args.batch_size=5
    args.learning_rate=0.001

    data_loader=DataLoader(graph)
    args.num_of_nodes=data_loader.num_of_nodes
    model=LINE(args)
    with tf.compat.v1.Session() as sess:
        tf.compat.v1.global_variables_initializer().run()
        initial_embedding=sess.run(model.embedding)
        learning_rate=args.learning_rate
        for i in range(args.num_batches):
            u_i,u_j,label=data_loader.fetch_batch(batch_size=args.batch_size,K=args.
 ↪K)
            feed_dict={model.u_i:u_i,model.u_j:u_j,model.label:label,model.
 ↪learning_rate:learning_rate}
            loss,_=sess.run([model.loss,model.train_op],feed_dict=feed_dict)
        if i==args.num_batches-1:
            embedding=sess.run(model.embedding)
            normalized_embedding = embedding / np.linalg.norm(embedding, axis=1,␣
 ↪keepdims=True)
            return data_loader.embedding_mapping(normalized_embedding)
```

[153]:
```
#Patient-medicine embedding dict
embedding_dict = train(G)
```

[154]:
```
keys = embedding_dict.keys()
patient_embedding = {}
medicine_embedding = {}
for i in list(keys):
    if str(i).isnumeric():
        patient_embedding[str(i)] = embedding_dict[i]
    else:
        medicine_embedding[i] = embedding_dict[i]
```

[163]:
```
# TransD embedding for the drugs of medicine knowledge graph
medicine_data = pd.read_csv ("./pykg2vec/examples/data/embeddings/transd/
 ↪ent_embedding.tsv", sep = '\t', header=None)
label_data = pd.read_csv ("./pykg2vec/examples/data/embeddings/transd/ent_labels.
 ↪tsv", sep = '\t', header=None)
medicine_data = medicine_data.to_numpy()
label_data = label_data.to_numpy()
medicine_kg_embedding = {}
for index, label in enumerate(label_data):
    medicine_kg_embedding[label[0]] = medicine_data[index]

# TransD embedding for the relations in medicine knowledge graph
med_rel_data = pd.read_csv ("./pykg2vec/examples/data/embeddings/transd/
 ↪rel_embedding.tsv", sep = '\t', header=None)
med_label_data = pd.read_csv ("./pykg2vec/examples/data/embeddings/transd/rel_labels.
 ↪tsv", sep = '\t', header=None, encoding='unicode_escape')
med_rel_data = med_rel_data.to_numpy()
med_label_data = med_label_data.to_numpy()
med_rel_embedding = {}
for index, label in enumerate(med_label_data):
    med_rel_embedding[label[0]] = med_rel_data[index]
```

[306]:
```
# To construct disease knowledge graph, use the ICD9 Ontology dataset
disease_df = pd.read_csv('ICD9CM.csv')
disease_df = disease_df[['Class ID','Parents']]
disease_df['type'] = 'childOf'
disease_df = disease_df.rename(columns={"Class ID": "target", "Parents": "source"})
disease_df = disease_df[['target','type','source']]
disease_df['source'] = disease_df.source.apply(lambda x: str(x).split('/')[-1].
 ↪replace('.', ''))
disease_df['target'] = disease_df.target.apply(lambda x: str(x).split('/')[-1].
 ↪replace('.', ''))
# Split into train, test and valid to calculate the embedding
X_train, X_test_val = train_test_split(disease_df, test_size=0.6)
X_test, X_val = train_test_split(X_test_val, test_size=0.5)
X_train.to_csv('./disease-train.csv',sep='\t')
X_val.to_csv('./disease-valid.csv',sep='\t')
X_test.to_csv('./disease-test.csv',sep='\t')
```

[157]:
```
# Construct patient disease bipartite network
disease_patient_df = pd.read_csv('DIAGNOSES_ICD.csv')
disease_patient_df = disease_patient_df[['subject_id','icd9_code']]
G1 = nx.Graph()
G1.add_nodes_from(disease_patient_df['subject_id'], bipartite=0)
G1.add_nodes_from(disease_patient_df['icd9_code'], bipartite=1)
G1.add_weighted_edges_from(
    [(row['subject_id'], row['icd9_code'], 1) for idx, row in disease_patient_df.
 ↪iterrows()], weight='weight')
```

[158]:
```
disease_embedding_dict = train(G1)
keys = disease_embedding_dict.keys()
disease_embedding = {}

for i in list(keys):
    if str(i).isnumeric():
        lists = []
        if str(i) in patient_embedding:
            lists.append(patient_embedding[str(i)])
        lists.append(disease_embedding_dict[i])
        # Combine the patient embedding from p-m bipartite network and p-d bipartite␣
 ↪network
        patient_embedding[str(i)] = list(map(sum, zip(*lists)))
    else:
        disease_embedding[i] = disease_embedding_dict[i]
```

[308]:
```
# Combine the medicine embedding from p-m bipartite network and medicine knowledge␣
 ↪graph
for i in medicine_kg_embedding:
    if i in medicine_embedding:
        lists = []
        lists.append(medicine_embedding[i])
        lists.append(medicine_kg_embedding[i])
        medicine_kg_embedding[i] = list(map(sum, zip(*lists)))
```

[160]:
```
# Get the disease embedding from disease knowledge graph and combine with embedding␣
 ↪from p-d knowledge graph
disease_data = pd.read_csv ("./pykg2vec/examples/disease-kg/embeddings/transd/
 ↪ent_embedding.tsv", sep = '\t', header=None)
disease_label_data = pd.read_csv ("./pykg2vec/examples/disease-kg/embeddings/transd/
 ↪ent_labels.tsv", sep = '\t', header=None)
disease_data = disease_data.to_numpy()
disease_label_data = disease_label_data.to_numpy()
medicine_kg_embedding = {}
for index, label in enumerate(disease_label_data):
    lists = []
    if label[0] in disease_embedding:
        lists.append(disease_embedding[label[0]])
    lists.append(disease_data[index])
    disease_embedding[label[0]] = list(map(sum, zip(*lists)))
```

[279]:
```
p = 10019
```

[290]:
```
sepsis_drug = ['Vancomycin','Ceftriaxone','Cefuroxime','Ceftin','Vancocin']
hypertension_drug = ['Lisinopril','Amlodipine','Norvasc','Carvedilol','Furosemide']
resp_failure_drug =␣
 ↪['Acetaminophen','Norepinephrine','Prednisone','Albuterol','Noxivent']
```

[281]:
```
patient_em = patient_embedding[str(p)]
```

[297]:
```
def dot_product(arr1, arr2):
    prod = 0
    for i, v in enumerate(arr1):
        prod = prod + (arr1[i] * arr2[i])
    return prod

def l1_norm(arr):
    norm = 0
    for i in arr:
        norm = norm + abs(i)
    return norm

def sum_arr(arr1, arr2):
    return_arr = []
    for i, v in enumerate(arr1):
        return_arr.append(arr1[i] + arr2[i])
    return return_arr

def diff_arr(arr1, arr2):
    return_arr = []
    for i, v in enumerate(arr1):
        return_arr.append(arr1[i] - arr2[i])
    return return_arr

def scalar_product(scalar, arr):
    return_arr = []
    for i in arr:
        return_arr.append(i * scalar)
    return return_arr
```

[311]:
```
def calculate_ddi(patient_em, arr1, arr2):
    n = 5
    res_scores = {}

    for index, value in enumerate(arr1):
        if value in list(medicine_kg_embedding.keys()):
            res = dot_product(patient_em, medicine_kg_embedding[value])
            scores[value] = res

    for i in range(0, n):
        value = arr1[i]
        if value in scores:
            for j in arr2:
                ddi = 0
                relation_df = medicine_kg_df[(medicine_kg_df['source'] == value) &␣
↪(medicine_kg_df['target'] == j)]
                relation = relation_df['edge'].unique()
                if len(relation) > 0:
                    relation = relation[0]
                    ddi = ddi +␣
↪l1_norm(diff_arr(sum_arr(medicine_kg_embedding[value],␣
↪med_rel_embedding[relation]), medicine_kg_embedding[j]))
            res_scores[value] = scores[value] - ddi
    return res_scores
```

[293]:
```
sepsis_ddi = calculate_ddi(patient_em, sepsis_drug, hypertension_drug)
sepsis_ddi
```

[293]:
```
{'Vancomycin': -554.9162230729705,
 'Ceftriaxone': -557.8339226240262,
 'Cefuroxime': -558.7119698473205}
```

[294]:
```
hypertension_ddi = calculate_ddi(patient_em, hypertension_drug, sepsis_drug)
hypertension_ddi
```

[294]:
```
{'Lisinopril': -14.056283833129838,
 'Amlodipine': 3.3913091023758932,
 'Carvedilol': -4.981197779652382,
 'Furosemide': 12.222794158188186}
```

[296]:
```
resp_failure_ddi = calculate_ddi(patient_em, resp_failure_drug, sepsis_drug)
resp_failure_ddi
```

[296]:
```
{'Acetaminophen': 34.319806024994,
 'Norepinephrine': -4.349440260140723,
 'Prednisone': -6.313434044598619}
```

[309]:
```
# 99592 - sepsis
# 4019 - unspecified essential hypertension

diseases = ['99592', '4019']
for i, val in enumerate(diseases):
    d = disease_embedding[val]
    res_scalar = scalar_product(math.exp(-(i+1)), d)
    if i == 0:
        new_patient_embedding = res_scalar
    else:
        new_patient_embedding = sum_arr(new_patient_embedding, res_scalar)
print(new_patient_embedding)
```

```
[0.22713131428177216, -0.06436416200443511, -0.06797525378304467,
0.11078386062666923, 0.08790227845927051, 0.030545909622186413,
0.1489080885635613, -0.09045192235260764, -0.12998357313164038,
-0.06548483085180652, -0.14461861927186187, 0.04129065622747197,
-0.0033810655376894025, -0.06675221620062702, -0.08474981671774566,
-0.017934030150872325, -0.05666303840875047, 0.12161384445041852,
0.059725237774324244, 0.30432979887447953, 0.09906956352084226,
-0.23363533089997313, -0.002737359841358377, -0.10742571750330182,
-0.12811973155023276, 0.13554820451096083, -0.007797355628315655,
-0.08919129111385406, 0.09034248716057855, -0.04636084356218427,
0.10136227174155837, 0.018091533589574334, 0.047255239550534844,
-0.06489889817160005, 0.027954206776750957, -0.013842769886809839,
0.32765129684885635, -0.055574395194754406, -0.2457427589361605,
-0.05581639173345987, -0.07683065207616775, -0.19231008188434578,
0.355714113452659, 0.3364620643825719, 0.11005410031536546, 0.06878400394254308,
0.01804434809041995, 0.002399543654860902, 0.006558955704588594,
-0.05302520151134536]
```

[312]:
```
sepsis_ddi = calculate_ddi(new_patient_embedding, sepsis_drug, hypertension_drug)
sepsis_ddi
```

[312]:
```
{'Vancomycin': -548.910495202641,
 'Ceftriaxone': -545.8999233213062,
 'Cefuroxime': -561.4798174361738}
```

# 9 Conclusion and Further Work

Further work of exploring different embedding techniques and measuring the performance against the current model is in progress. Also, in future, the prediction accuracy can be measured using a Jaccard coefficient which in turn can be used to compare the efficiency and accuracy of the different models. The part of finding useful medicine given a disease is still in progress which can be automated by implementing link prediction between disease knowledge graph and medicine knowledge graph.

Thus, a drug recommendation system has been implemented using TransD embedding and LINE embedding between knowledge graph and bipartite graph which can do a safe drug recommendation after analysing the drug interactions.

# References


[1] Susannah Fox Maeve Duggan. "Health Online 2013". In: *Pew Research Center* (2013).

[2] Fang Gong et al. "SMR: Medical Knowledge Graph Embedding for Safe Medicine Recommendation". In: (2020). arXiv: 1710.05980 [cs.IR].

[3] "HIQA recommends promoting a national strategic approach to reduce the amount of medication errors in public acute hospitals". In: *HIQA* (2018).

[4] Knowledge Graph. *Knowledge graph — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-October-2021]. URL: https://en.wikipedia.org/wiki/Knowledge_graph.

[5] Knowledge Graph Embedding. *Knowledge graph embedding — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-October-2021]. URL: https://en.wikipedia.org/wiki/Knowledge_graph_embedding#Pure_translational_models.

[6] Zhiyuan Liu Yankai Lin et al. "Learning entity and relation embeddings for knowledge graph completion". In: *Proc. 29th AAAI Conf. Artif. Intell* (2020).

[7] Guoliang Ji et al. "Knowledge Graph Embedding via Dynamic Mapping Matrix". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 687–696. DOI: 10.3115/v1/P15-1067. URL: https://aclanthology.org/P15-1067.

[8] Jian Tang et al. "LINE". In: *Proceedings of the 24th International Conference on World Wide Web* (May 2015). DOI: 10.1145/2736277.2741093. URL: http://dx.doi.org/10.1145/2736277.2741093.

[9] Wishart DS, Feunang YD, Guo AC, Lo EJ, Marcu A, Grant JR, Sajed T, Johnson D, Li C, Sayeeda Z, Assempour N, Iynkkaran I, Liu Y, Maciejewski A, Gale N, Wilson A, Chin L, Cummings R, Le D, Pon A, Knox C, Wilson M. DrugBank 5.0: a major update to the DrugBank database for 2018. Nucleic Acids Res. 2017 Nov 8. doi: 10.1093/nar/gkx1037.

[10] A. E. Johnson, T. J. Pollard, L. Shen, L.-w. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, R. G. Mark, Mimic-iii, a freely accessible critical care database, Scientific Data 3.

[11] L. M. Schriml, C. Arze, S. Nadendla, Y.-W. W. Chang, M. Mazaitis, V. Felix, G. Feng, W. A. Kibbe, Disease ontology: a backbone for disease semantic integration, Nucleic Acids Research 40 (D1) (2011) D940–D946.

[12] Shih Yuan Yu et al. "Pykg2vec: A Python Library for Knowledge Graph Embedding". In: *arXiv preprint arXiv:1906.04239* (2019).

[13] https://github.com/ninoxjy/graph-embedding.