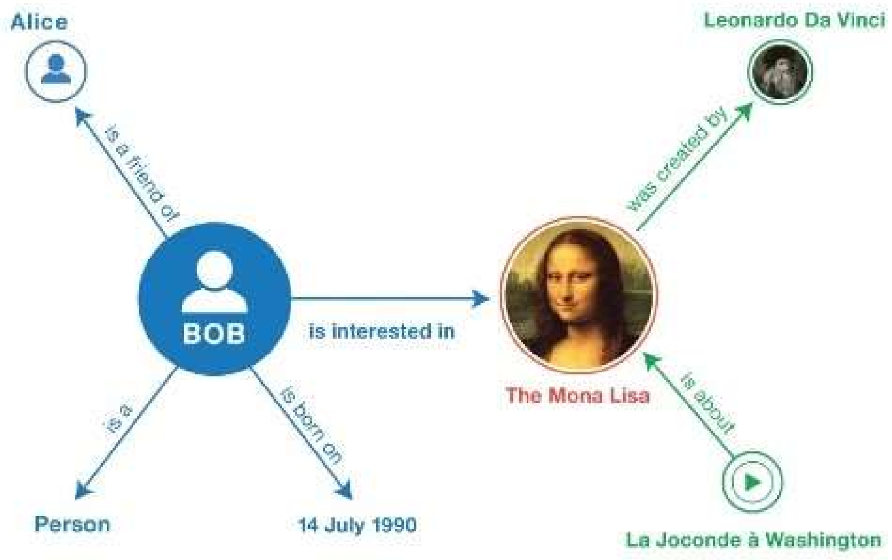
Introduction

What is a knowledge graph ?

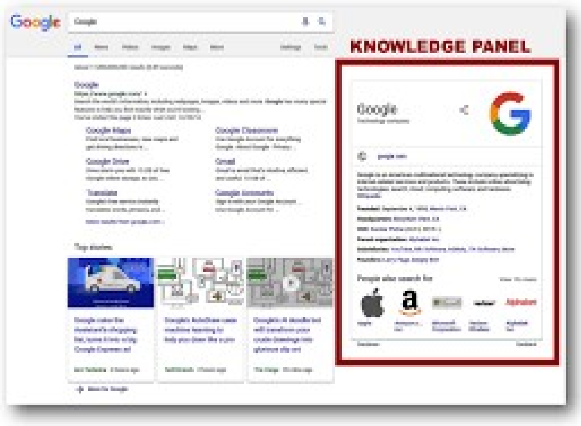
A knowledge graph is a way to represent information and the relations that exist between its basic units. It is a graph with the entities (basic units of information) as the nodes and the edges between them as their relation.

Here is a small example :



Used by search engines

Knowledge graphs are used by search engines like Google to improve their search results. Information from Google's KG can be seen presented in the knowledge panel to the right of the screen:



As of October 2016, Google's KG was reported to contain over 70 billion facts

Applications to AI Systems

Most AI systems today fail to do well due to the lack of a ‘common sense’. They don’t have a general understanding of the world, like we humans do, to provide them with common sense. As an example – A self-driving car might stop for a snowman, thinking that it may cross the road; and NLP systems may fail to answer basic questions based on common sense logic like ‘If I put my socks in a drawer, will they still be there tomorrow ?’ or ‘How can you tell if a milk carton is full ?’. Most of the times, AI systems only have very specific domain knowledge.

So state of the art ML / AI techniques are now attempting to make this general knowledge available to ML models, to help them perform better. A KG is an excellent way to represent this external information. Nodes define entities and the edges between them define the relations. Such a KG can then be fed to the ML model.

Problem Context

The Dialog State Tracking Challenge (DSTC) was initially started to serve as a testbed for dialog state tracking task and was rebranded as Dialog System Technology Challenges in 2016 to provide a benchmark for evaluating dialog systems. The latest DSTC challenge, DSTC7, is divided into three different tracks. The present work addresses the Sentence Selection track, where we have to basically find the correct next utterance, given a partial conversation, from a set of candidates. This can be used to solve the ubuntu users’ queries.

Objectives

1. Select next utterance from a set if candidates
2. Select none if all are wrong
3. Learn to incorporate external knowledge sources

The baseline model for this task is the ESIM model (Enhanced Sequential Information Model)

This model will be provided with external knowledge and enhanced to the K-ESIM (Knowledge based ESIM) model. The external knowledge will be fed in the model in the form of a knowledge graph constructed from all the ubuntu man pages. The project addresses the question of how to actually build this knowledge graph

The Algorithm for knowledge graph construction

The programs, in order of their execution, are explained below

Full Entity Extraction

The following process is repeated for every man page:

First, the text from the ‘Name’ and ‘Description’ sections of the man page is found and stored in name_code and desc_code. If both are ‘None’ meaning that the file has no ‘Name’ and ‘Description’ sections, then the file is not a valid man page and no further processing is done.

1. Build the entity list
   The man page is first parsed using python’s beautiful soup package. Further entity search is done on the beautiful soup object thus created.
   a) Title entities: The title string is extracted from the name_code by doing an appropriate regex search. If this title string is ‘None’, then the title string is just extracted from the file name. This string is then split by comma to get the list of all titles present. We then append the section id to each title found just now to get the final title entities.
   b) All href words / phrases: Use soup.find_all(‘a’)
   c) Boldface words: Find all the words in the ‘Description’ section of the man page that are between <b> and </b> by doing a regex search
   d) Italics Words: Similar to part c. Find all words in the ‘Description’ section of the man page that are between <i> and </i> by doing a regex search
   e) Man page mentions: Find words that end with (0-9)
   f) System file mentions: Find words that have 2 or more parts like /{something}
   g) Words in all caps: Simple regex search
   h) All first letter capital mid sentence words
   i) List of proper nouns: First, the nltk package is used to split the text into sentences and words. Then chunking is used to search for a comma (<CC> in general) separated list of <NNP>s.
   j) Anchor entities: The ‘SEE ALSO’ section of the man page is searched for man page mentions

   The full entity list is built by taking a union of all the entities found above. The type of each entity is also stored in a type_dict.

Post processing is carried out on the entire entity list built above. Some steps are stop words removal, removing word if word(digit) is already present, changing ‘word (digit)’ to ‘word(digit)’, removing entities with length <= 1, etc. The type_dict is modified accordingly.

2. Holling
   The sentences and words extracted in a previous step (using nltk) are used for this. First, the maximum word length among all entities (say len) is calculated. Then a new entity set, holl_ent_set is derived from the original entity set by removing all the spaces in between entity words. The mapping from the new entity to the old original entity is stored in holl_index.
   For each sentence, we do the following:
   a) We take groups of words ranging from group size 1 upto len, and for each group, we concatenate all the words and find out if this new concatenated word occurs in the holl_ent_set. This can happen if and only if an entity has been found at that position. The original entity corresponding to the entity from holl_ent_set just found, and its position of occurrence are stored in the word_list
   b) At the end, we examine all the entities in the word_list one by one. For each entity, we store its POS tag composed context, as well as the word composed context in separate dictionaries (context_dict and context_word_dict) respectively.
   c) The word_list is now used to find pairs of entities that occur in some +- k window of each other (using the find_pairs function). The word list consists of entities sorted according to (position of occurrence, entity length) and this fact is exploited to make the algorithm fast.
3. Next the occurrence count for each entity is calculated, by summing up the counts of each type of context in the context dictionary for the entity. If the occurrence count for any entity turns out to be zero (meaning that the entity is present in the entity list but was not found in holling), it indicates that there is some special case that the code did not consider. Such entities are removed from the entity list to be safe
4. Now the extractions for this man page are written to 2 separate files, one that stores the properties of each extracted entity and the other that stores only the title and anchor entities. The name for the files is obtained by replacing the .html at the end of the downloaded file by .txt. The anchor entities’ file is stored in the ‘Anchors’ folder.

a) Write the properties of each entity to a file.
   The format followed is:
   Entity_name \t type \t context_dict \t co-occuring entities
b) Now the anchor and title entities are written to a separate file. For this purpose, the lists of title and anchor entities are recalculated because the post processing of the entity list might have removed / changed certain title and anchor entities.
   The format followed is:
   Title \t anchor_entities(links) for each title found

Merger

This program merges all the separate extraction files created from all the man pages to create a single set of entities. The output is stored in 2 files: One file (ent_index.txt) that stores the entity to index mapping, and the second file that stores the overall properties of merged entities (index_merged_entities.txt).

The algorithm is:

1. First the entity set is initialized to empty. The following procedure occurs for every man page extraction file that stores the entity properties for that man page – the file is read line by line and the entity name in that line is examined. If that name already exists in the entity set, then the properties for that entity (type, context_dict, co_occuring_entities) are merged with those already present in the entity set
2. The entity set is created. The program now assigns a unique index to each entity and writes it to a file. Basically, it just loops through the entity set and assigns the loop index variable's value to the entity at that stage in the loop
3. After this, the entity indices are read from the file just created. They are stored in a dictionary that gives the entity to index mapping
4. Finally, the merged properties of all the entities are written a separate file. Each entity name occurring in any property is replaced by its corresponding index while writing to the file

Knowledge Graph Maker

This program uses the merged entity list to extract relations to build the knowledge graph. Relations of each type are stored in a separate file. The different types of relations extracted are:

1a: The title of each man page is related to every other entity of the same man page by this relation. It is a one way relation, stored in the format entity_index_1   \t   entity_index_2. The title for each man page is extracted from the anchor file for that man page, which is then connected to all the entities in the extraction file for that man page

1b: Each entity is connected to every other entity that co-occurs with it in some +- k window. It is a one way relation, meaning that each relation in the ‘1b.txt’ file specifies a single directed edge between the first and second entity. Format is entity_index_1   \t   entity_index_2. This co-occurrence information is directly available in the ‘index_merged_extractions.txt’ file, and thus that file is just parsed once to extract these relations

2: These are the anchor links, i.e. each man page title is connected to other man page titles that occur in its ‘SEE ALSO’ section. This is also a one way relation and the storage format is entity_index_1   \t   entity_index_2. For this relation extraction, the anchor file of each man page is just parsed once to get the relations corresponding to that man page.

3: Entities that have a similar context_dict are connected by this relation. The context_dict for every entity is available in the ‘index_merged_extractions.txt’ file. Thus first, that file is read to get the context_dict for each entity. Then every pair of entities of entities are compared to see of they are related. Two entities will be related if and only if:
No of comment context for entity 1 >= threshold * Total no of contexts for entity 1 and No of comment context for entity 2 >= threshold * Total no of contexts for entity 2.
The no of common contexts for an entity is obtained by summing up the occurrence counts of each common context for that entity. Thus 2 entities that are compared will in general have different no of common contexts.
This is a 2 way relation, meaning that each line in the file actually represents an undirected edge between the entities. File format is entity_index_1   \t   entity_index_2

4: Entities that occur within a fixed window size are connected by the zip words that appear between them. These set of zip words are supposed to represent the relation between the entities. For this relation, each man page is parsed and the entities that occur within the fixed window size are noted. The extractions are stored in a file in the format:

entity_index_1   \t   relation \t   entity_index_2 where relation is the set of zip words that occur between entity1 and entity2

Zipf's law

This script finds and stores all the zip words from the entire ubuntu man page corpus, which are used to extract the type 4 relations (described earlier). First, the frequency of all the words in the corpus is found and stored. Then, all the sorted by their frequency into a list. Then the user has to specify the value of two indices: lim1 and lim2, and the zip word set will be the set of words from indices lim1 to lim2. Words having indices less than lim1 are much too frequent and hence ignored. Words with indices greater than lim2 are too rare to be of any importance.

Knowledge Graph Reader:

Reads the knowledge graph from secondary storage into primary memory