Antonio Norelli

Natural Language Processing *a.a. 2016/2017* *prof. Roberto Navigli*

Final project: The Knowledge Bot

CONTENTS

Antonio Norelli

Natural Language Processing *a.a. 2016/2017* *prof. Roberto Navigli*

Final project: The Knowledge Bot

Introduction

The goal of the project is to build a Telegram chatbot able to answer to questions about general culture (querying) and able to learn by asking questions itself (enriching). This result is achieved using a knowledge database. Every entry in the dataset is composed by a couple of concepts, the linking relation and a query-answer couple that expresses this information. Breaking down the problem:

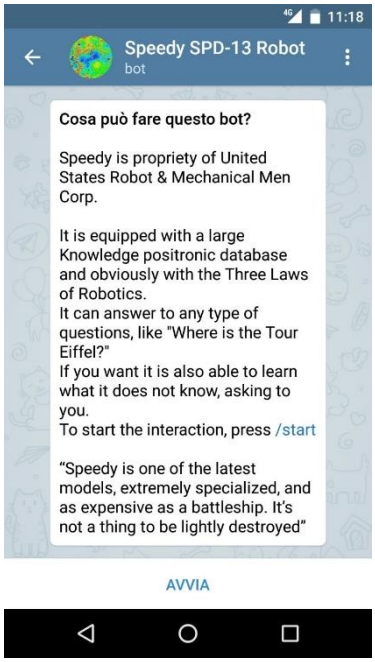
- In the querying modality the bot should be able to detect the key concept in the question, detect the relation associated to the question and search in the dataset an appropriate answer.
- In the enriching modality the bot should be able to find a concept-relation couple that makes sense and absent in the dataset, build a proper question from it, collect the answer from the user, detect the key concept in the answer of the user and store it, in order to update the dataset.

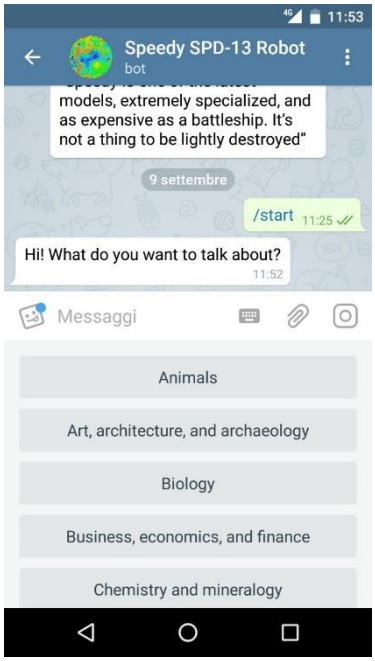
Additionally, the chatbot should be provided with a basic human computer interface, composed by a proper design of the conversations and eventually by the usage of the Telegram chatbot perks (commands, custom keyboards, text formatting...).

Human computer interface

In order to simplify and make more straightforward the interaction with the user, a gold rule for chatbots is to guide the conversation. So, taking this design choice, the chatbot begins to ask which is the preferred domain of knowledge the user want to talk about, then it asks about the modality (querying or enriching) and finally according to the modality it waits for a question or asks itself a first question. A custom keyboard is implemented for all the choices to makes them faster and simplify the management of the answers. As in the majority of chatbots, the bot messages are in natural language, this makes the user interaction very natural and simple. The only command implemented by the chatbot is */start*, that is used to begin the interaction or to reset it if the user wants to change domain or modality.

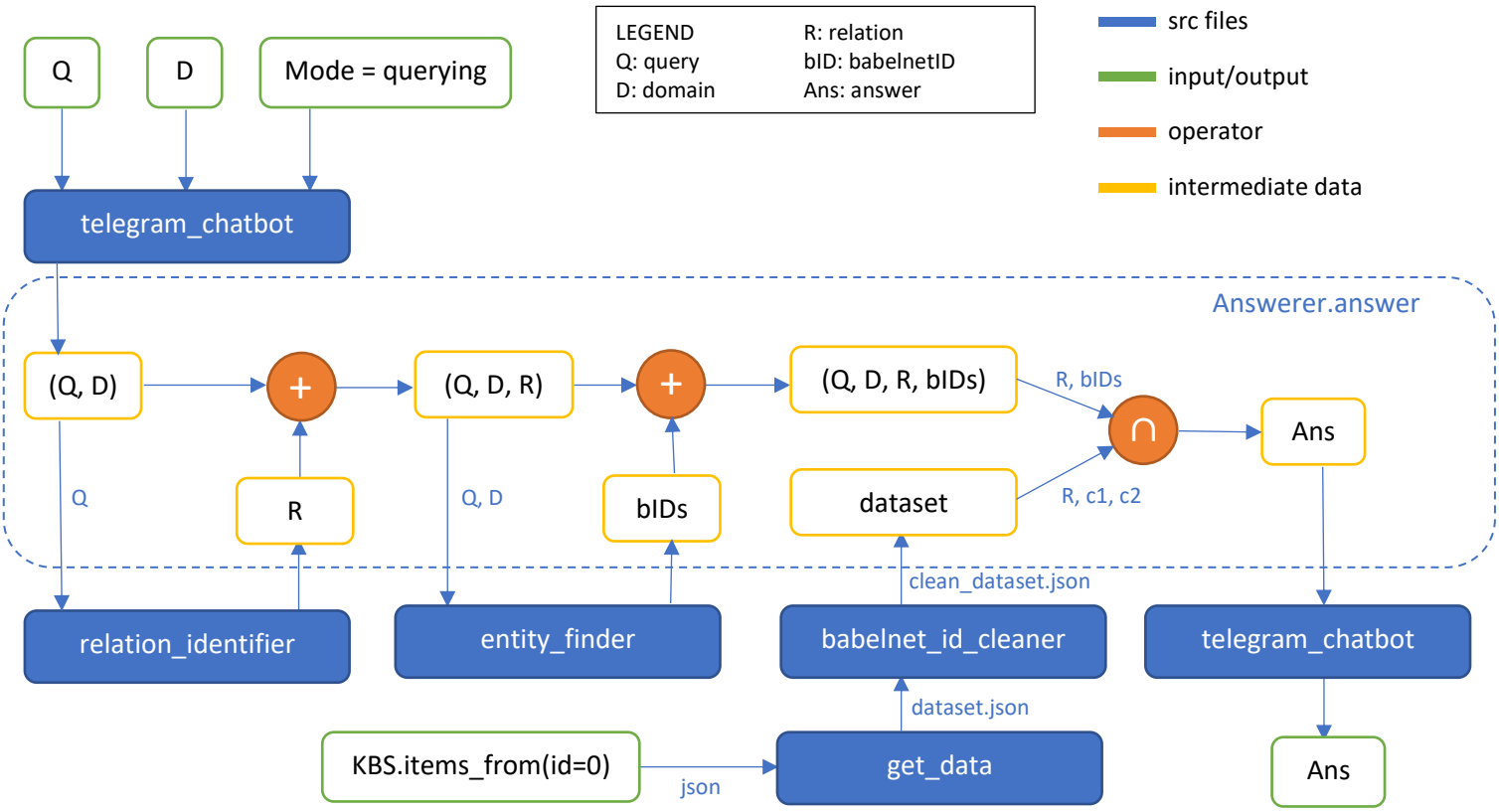
The following screenshots show a standard querying interaction.





Querying modality

The software architecture diagram in the querying modality is the following:



`telegram_chatbot.py` gets the last user messages from the Telegram API, in the querying modality it collects the domain and the user query and calls the `answer` method of the class `Answer.py`. `answer` sends the query to `relation_identifier.py`, receiving the predicted relation, then sends the query and the domain to `entity_finder.py`, receiving the babelnetIDs of the concepts in the query, then searches in the dataset an entry with those babelnetIDs and the correct relation, if such an entry is found it returns the answer to `telegram_chatbot.py` that interacts with the Telegram API to communicate the answer to the user. The raw dataset is prepared by `get_data.py` that acquires all the data from the KBS server and then standardized by `babelnet_id_cleaner.py`.

Relation identifier - input: Q, output: R

The representation used for the queries is a n-gram model (unigrams and bigrams), while the classifier used is a support vector machine with a linear kernel.

The training data is in `patterns_num.txt`, the mapping between relationID and relation is in `relations.txt`. `patterns_num.txt` was obtained merging all the `patterns.tsv` provided, excluding the ones without a correct format and the ones with wrong query-relation entries.

The n-gram representation is chosen because the main discriminant in the classification of a query is simply the presence of certain words, like "where" (PLACE), "when" (TIME), "color" (COLORPATTERN), "made of" (MATERIAL). Unigrams and bigrams seem sufficient to detect all these keywords since a test with also trigrams showed no improvement. Considering the reduced size of the dataset and the expected simplicity of the decision boundary, the chosen classifier is a support vector machine.

In order to optimize the hyperparameters of the chosen classifier, the training data is split in training, dev and test set and a grid search is performed. Results of the grid search are in the below boxes, the final performance of the classifier is slightly better than the one presented here since during later tests some

wrong samples in `patterns_num.txt` are found and deleted. A complete performance analysis is done in the last section.

```
RELATION IDENTIFIER
Training, Dev, Test size: 0.6, 0.2, 0.2

Grid scores on development set (f1 score):

0.664 (+/-0.005) for {'clf__C': 0.1, 'clf__kernel': 'linear'}
0.772 (+/-0.034) for {'clf__C': 1, 'clf__kernel': 'linear'}
0.772 (+/-0.034) for {'clf__C': 10, 'clf__kernel': 'linear'}
0.772 (+/-0.034) for {'clf__C': 100, 'clf__kernel': 'linear'}
0.156 (+/-0.011) for {'clf__C': 0.1, 'clf__gamma': 0.001, 'clf__kernel': 'rbf'}
0.156 (+/-0.011) for {'clf__C': 0.1, 'clf__gamma': 0.01, 'clf__kernel': 'rbf'}
0.156 (+/-0.011) for {'clf__C': 0.1, 'clf__gamma': 0.1, 'clf__kernel': 'rbf'}
0.156 (+/-0.011) for {'clf__C': 1, 'clf__gamma': 0.001, 'clf__kernel': 'rbf'}
0.164 (+/-0.012) for {'clf__C': 1, 'clf__gamma': 0.01, 'clf__kernel': 'rbf'}
0.660 (+/-0.008) for {'clf__C': 1, 'clf__gamma': 0.1, 'clf__kernel': 'rbf'}
0.164 (+/-0.012) for {'clf__C': 10, 'clf__gamma': 0.001, 'clf__kernel': 'rbf'}
0.752 (+/-0.017) for {'clf__C': 10, 'clf__gamma': 0.01, 'clf__kernel': 'rbf'}
0.744 (+/-0.025) for {'clf__C': 10, 'clf__gamma': 0.1, 'clf__kernel': 'rbf'}
0.764 (+/-0.017) for {'clf__C': 100, 'clf__gamma': 0.001, 'clf__kernel': 'rbf'}
0.772 (+/-0.034) for {'clf__C': 100, 'clf__gamma': 0.01, 'clf__kernel': 'rbf'}
0.744 (+/-0.025) for {'clf__C': 100, 'clf__gamma': 0.1, 'clf__kernel': 'rbf'}
0.152 (+/-0.013) for {'clf__C': 0.1, 'clf__degree': 2, 'clf__kernel': 'poly'}
0.096 (+/-0.010) for {'clf__C': 0.1, 'clf__degree': 3, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 0.1, 'clf__degree': 4, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 0.1, 'clf__degree': 5, 'clf__kernel': 'poly'}
0.152 (+/-0.013) for {'clf__C': 1, 'clf__degree': 2, 'clf__kernel': 'poly'}
0.096 (+/-0.010) for {'clf__C': 1, 'clf__degree': 3, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 1, 'clf__degree': 4, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 1, 'clf__degree': 5, 'clf__kernel': 'poly'}
0.152 (+/-0.013) for {'clf__C': 10, 'clf__degree': 2, 'clf__kernel': 'poly'}
0.096 (+/-0.010) for {'clf__C': 10, 'clf__degree': 3, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 10, 'clf__degree': 4, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 10, 'clf__degree': 5, 'clf__kernel': 'poly'}
0.152 (+/-0.013) for {'clf__C': 100, 'clf__degree': 2, 'clf__kernel': 'poly'}
0.096 (+/-0.010) for {'clf__C': 100, 'clf__degree': 3, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 100, 'clf__degree': 4, 'clf__kernel': 'poly'}
0.092 (+/-0.007) for {'clf__C': 100, 'clf__degree': 5, 'clf__kernel': 'poly'}
```

```
Best parameters set found on development set:
C=1, kernel='linear'

Wrong classifications:
Which material  contains? 2 8
Is  increased or decreased? 5 8
When  can be used? 16 7
Is it possible to confuse  with ? 11 8
Was  at the epoch of ? 16 4
Was  composed by ? 8 2
Is  the use of  ? 6 12
What is the specialization of ? 7 1
Is  used for  ? 12 6
are  and  similar ? 11 8
What is the class of  ? 4 1
What is the category of ? 4 1
Is  used for ? 12 6
There was  in ? 2 16
              precision    recall  f1-score   support

           1       0.40      1.00      0.57         2
           2       0.75      0.60      0.67         5
           3       1.00      1.00      1.00         3
           4       0.50      0.33      0.40         3
           5       1.00      0.75      0.86         4
           6       0.60      0.75      0.67         4
           7       0.80      0.80      0.80         5
           8       0.50      0.80      0.62         5
           9       1.00      1.00      1.00         3
          10       1.00      1.00      1.00         5
          11       1.00      0.60      0.75         5
          12       0.67      0.50      0.57         4
          13       1.00      1.00      1.00         4
          14       1.00      1.00      1.00         5
          15       1.00      1.00      1.00         1
          16       0.75      0.60      0.67         5

avg / total       0.82      0.78      0.78        63
```

Note that a standard query contains the words that express the entities, differently from the query patterns that compose the training set, but those words do not affect the prediction since the vocabulary used for the vectorization through the n-gram model is the one of the training. This means that this classifier is capable to look only at the query pattern to detect the relation, most of the time this is sufficient but questions where the relation is mainly expressed by the entities (e.g. “Are soccer pitches rectangular?”) are often misclassified. This design choice makes the classifier simple and fast but weak with some type of questions, see the last section for some examples.

Entity finder - input: Q, D, output: bIDs

The entities in the query are detected using two different approaches, each one with different strengths and weaknesses. The first approach uses Babelfy while the second manually search the entities considering the POS tags and the dependency grammar tags of the words in the query and then uses Babelnet to obtain the corresponding babelnetID. The management of the two options is in the function `entity_finder` and controlled by the variable `spacy_dis`.

Babelfy entity finder

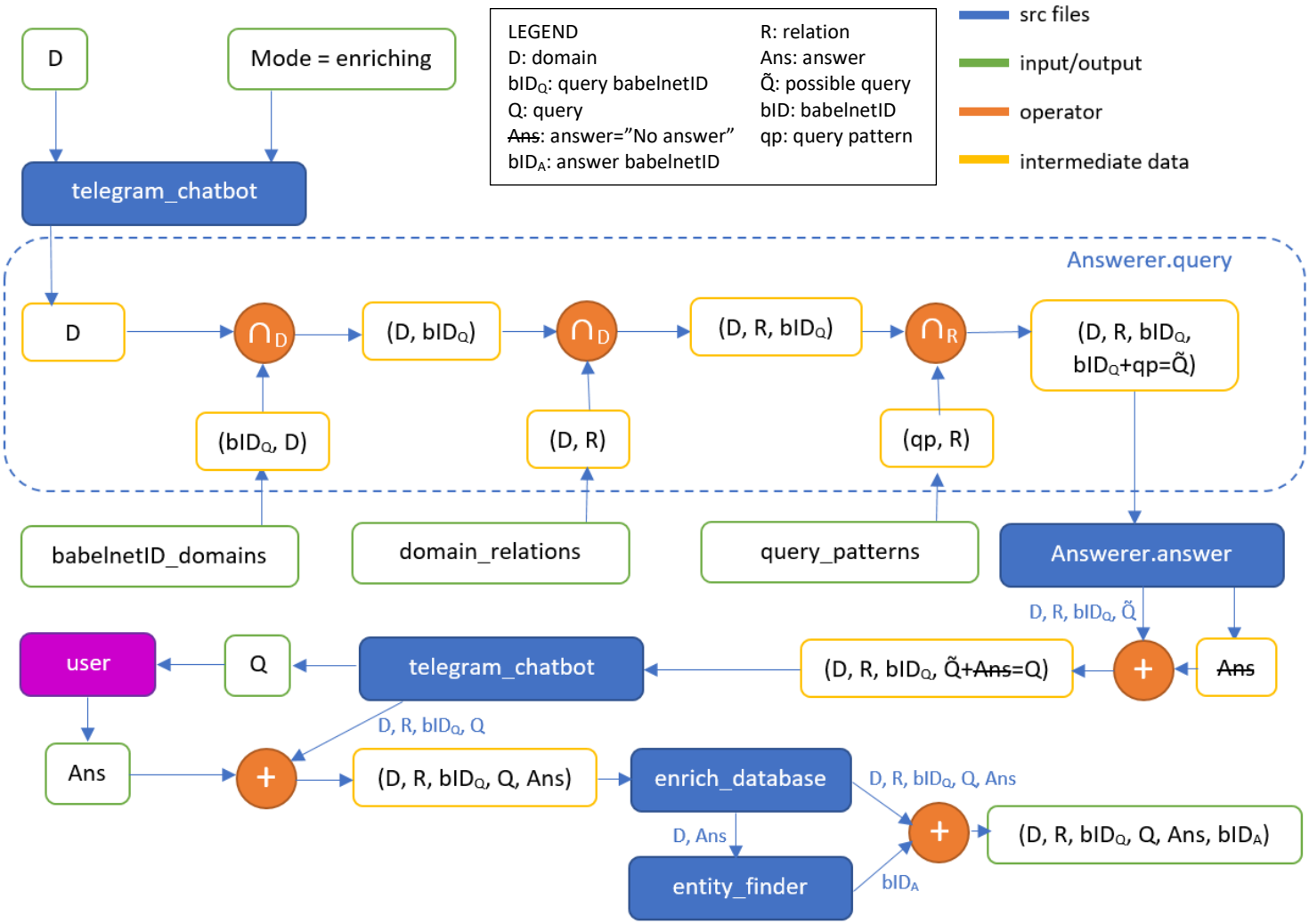
The input text is passed to the Babelfy API through the `disambiguate` command. The parameter `matching` is set to PARTIAL_MATCHING to gather all possible entities. The result is a json list of the found entities that includes the babelnetID of each entity, all these bIDs are collected. If one or more entities overlap, only the longest one is maintained (e.g. in the query “Where is the Statue of Liberty?”, the entities “Statue” and “Liberty” are discarded while “Statue of Liberty” is maintained). This method is fast and cheap in babelcoins, since a complete disambiguation of a text costs a single Babelcoin, on the other hand the disambiguation of Babelfy returns all the entities and not only the subject or the object of the query, so when the query is long and complex we may collect not interesting entities. Note that if a chunk of words can be associated to more than one entity, the disambiguation is performed directly by Babelfy, without the usage of the domain information.

Spacy entity finder

In this alternative method the input text is analysed with the spaCy NLP library that provides the POS tags and the dependency tree of the sentence. The entity candidates are the words that compose the subtrees that contain a subject or an object in which the subject or object is a noun, a proper noun or a number. The articles are removed from the chunk of words and then it is sent to the Babelnet API through the `getSenses` command, returning the correspondent babelnetID. If there is more than one possible concept, the first one is returned. If a domain is specified it is returned the first correspondent to the domain. The analysis of the dependency tree and the usage of the domain information on one hand make this approach more consistent but on the other hand it pays in the disambiguation task, in speed and in babelcoins due to the multiple calls at the Babelnet API.

Enriching modality

The software architecture diagram in the enriching modality is the following:



`telegram_chatbot.py` gets the last user messages from the Telegram API, in the enriching modality it collects only the domain and calls the `query` method of the class `Answer.py`. `query` searches in `babeldomains_babelnet` a bID of the chosen domain, then chooses randomly a relation associated to the domain looking at `domain_relations`, then chooses randomly a query pattern for the chosen relation and builds the possible query with the concept encoded in the babelnetID and the query pattern. The query candidate is asked to `Answerer.answer`, if the returned answer is "Sorry, I have not an answer for this question" we know that the chosen couple bID-relation is not in the dataset and the question is well defined, now the query candidate becomes the final query. This is sent to `telegram_chatbot` that sends it to the user, finally the answer of the user is collected and if it contains an entity the dataset is enriched.

Quantitative evaluation of the performance

What does it mean evaluating the performance of such a chatbot? The answer is not trivial, intuitively we evaluate a chatbot according to its answers, but those answers are the result of a complex process. What we intuitively consider a poor answer, such as "Sorry but I have not an answer for this question", may hide an appropriate prediction of all the models and simply be the consequence of a failure search in the database.

So what we really want to evaluate are the core models behind the chatbot, the relation identifier and the entity_finder, but here another question emerges, what is the test set? The ideal test set should come from user experience but this goes beyond the aims of this project. Therefore, the test set come from the Knowledge Base Server itself, that however hardly can be considered a gold standard. How much a poor performance of the relation identifier or of the entity finder can be attributed to a bad choice of the models and how much to corrupt entries in the KBS?

With this in mind a pre-evaluation of the consistency of the dataset is performed while the presented evaluation choices for the models try to minimize the influence of the corrupt entries in the KBS.

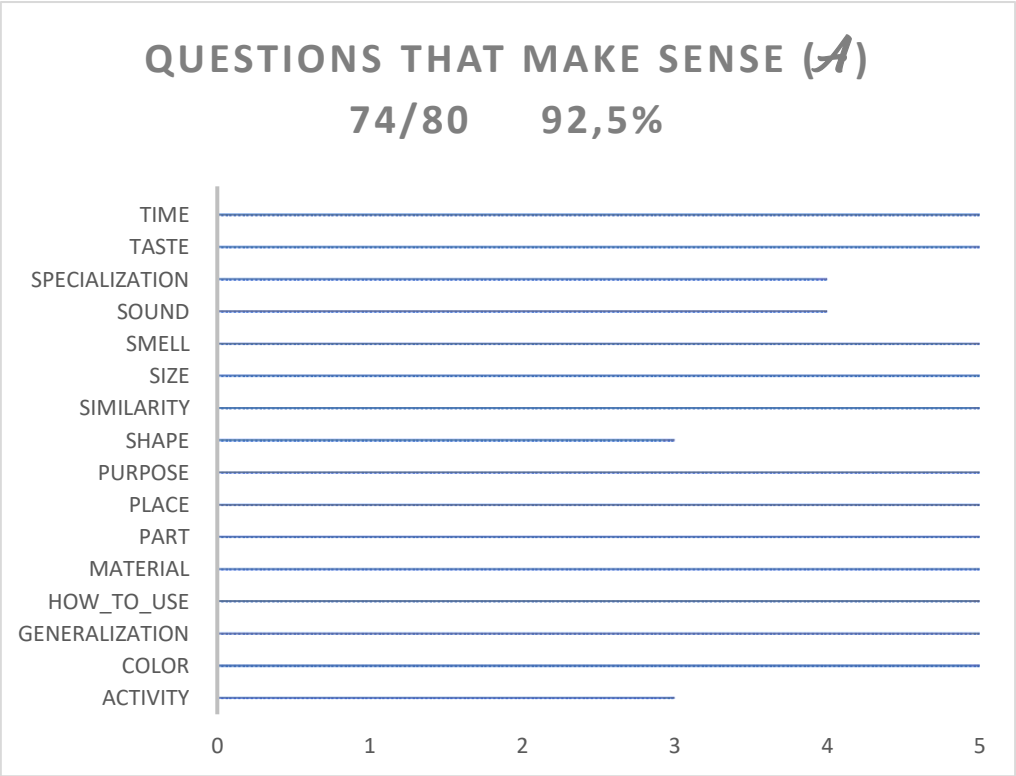
Dataset consistency

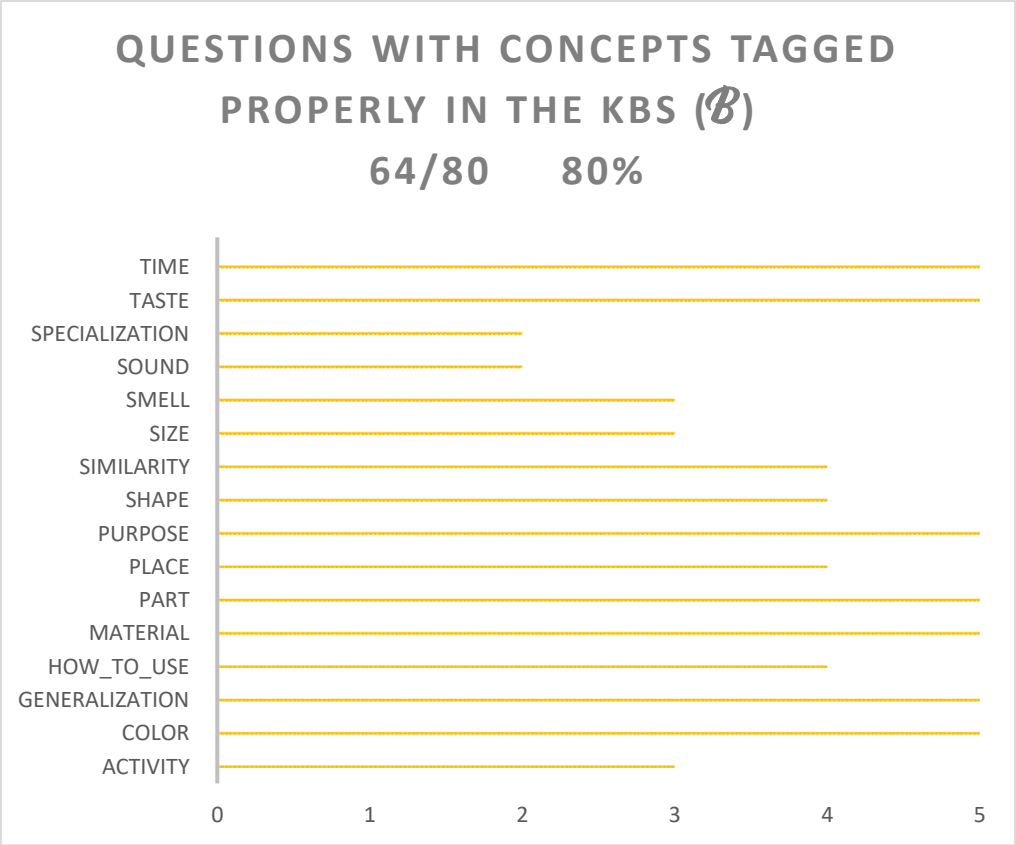
For the evaluation of the dataset and the evaluation of the entity finder is used a sample of 80 entries of the dataset, 5 for each relation. The only possibility is to do a handmade evaluation (that is why 80 samples and no more). For each entry is put a label 1 if the question makes sense, 0 if the opposite, and a label 1 if the concepts in the question are correctly tagged in c1 and/or c2, 0 if the opposite.

An example of question that makes sense is "What is the smell of success?"

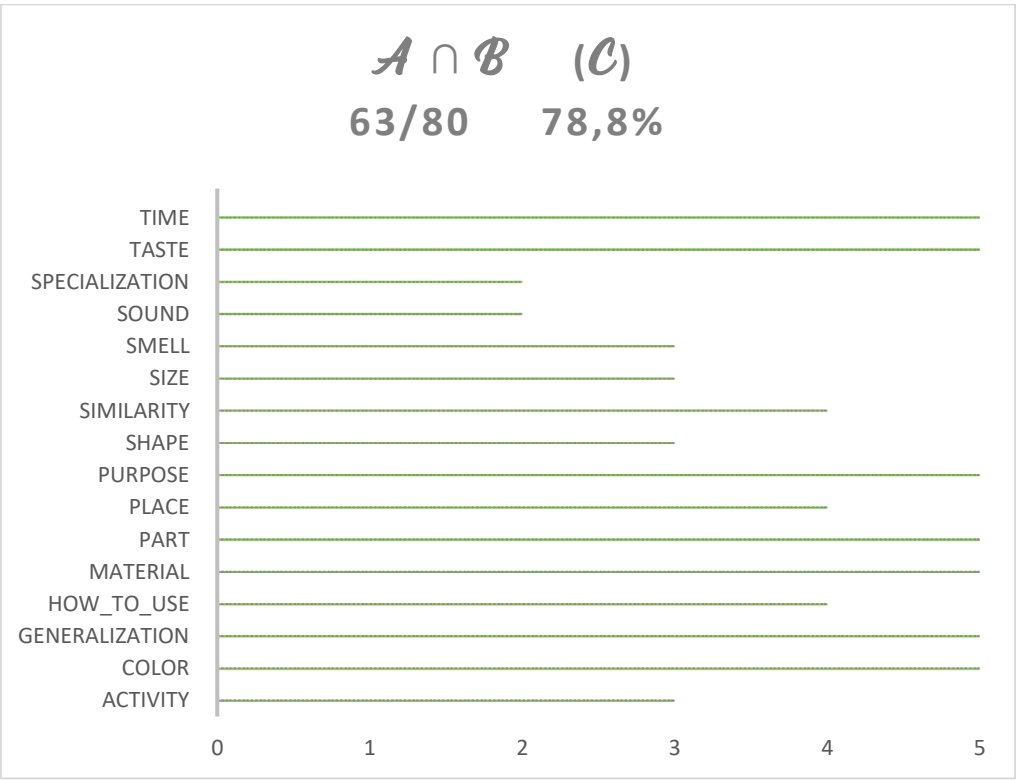
An example of question that makes no sense is "What kind of object is seen as?"

All the labels are in the *dataset* sheet in *stats.xlsx* in the output folder.





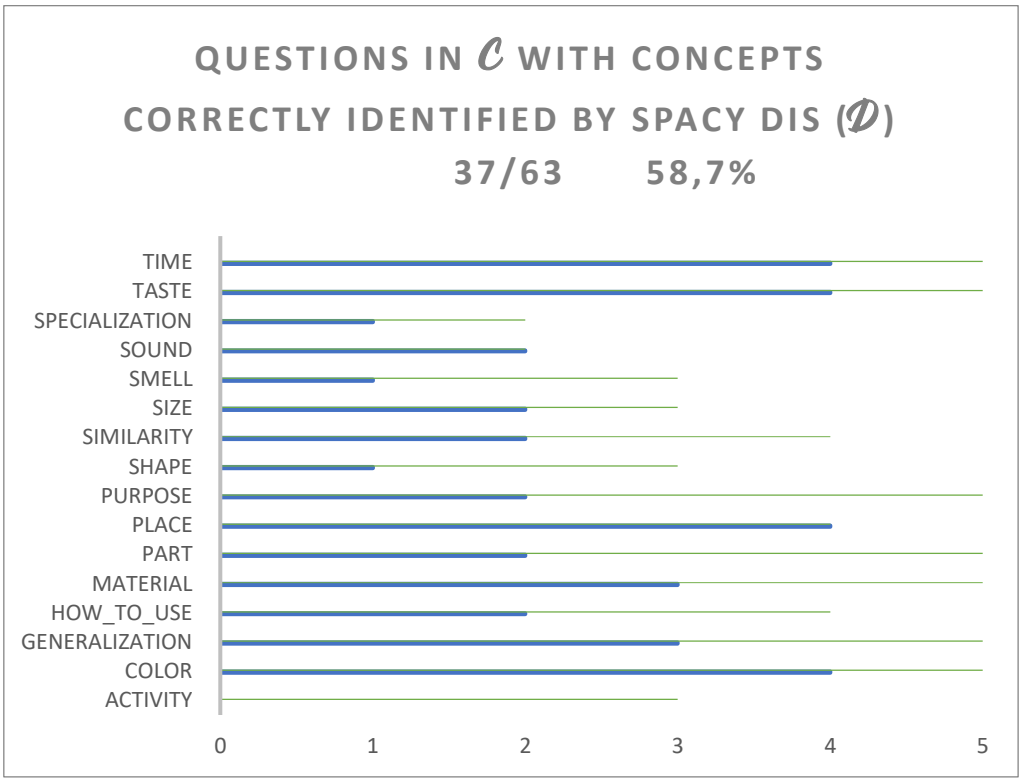
So overall, we have 63 valid entries over 80. Less than 80%.

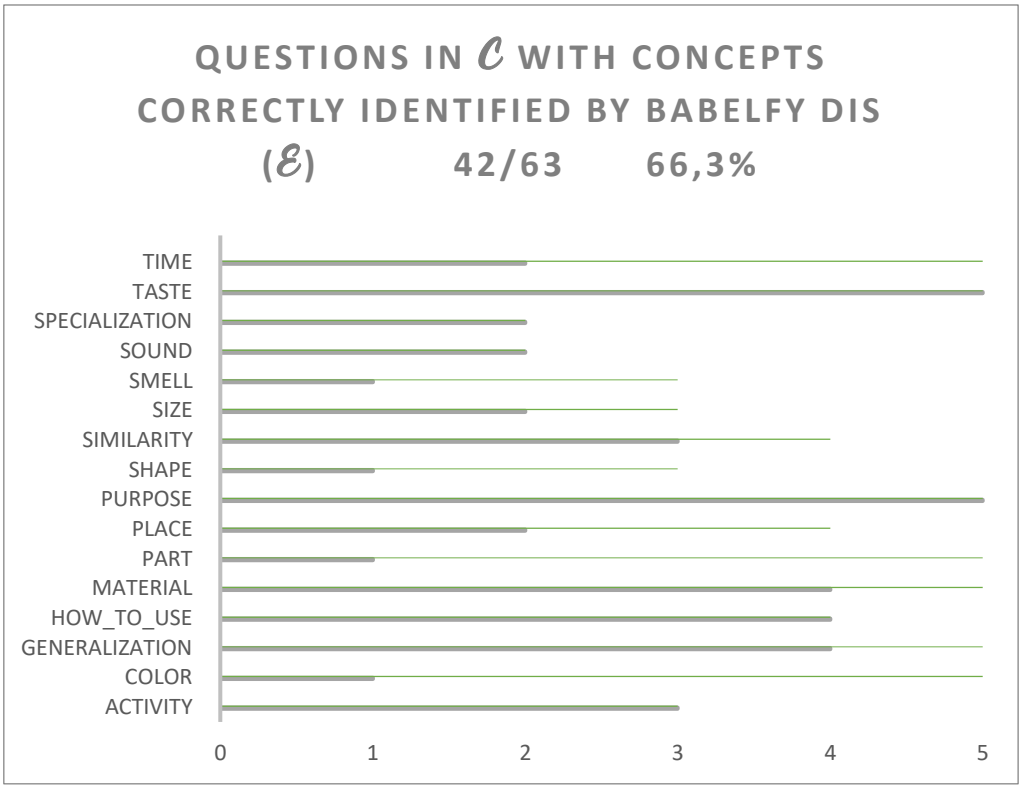


Entity finder

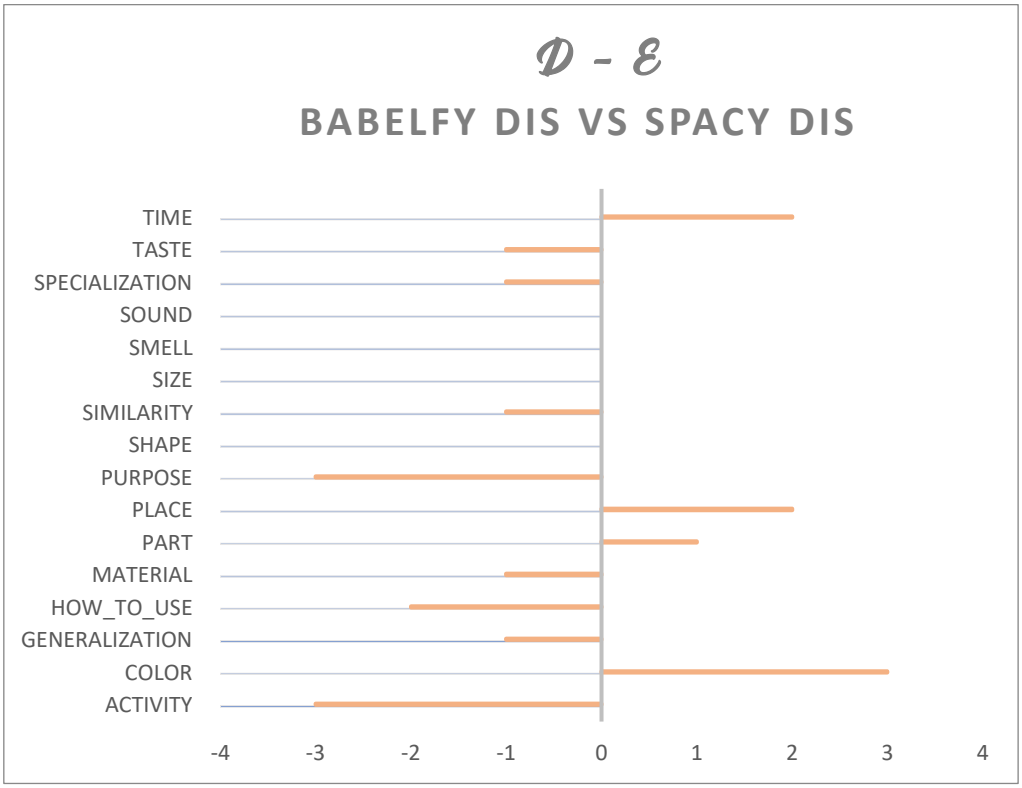
The entity finder performance evaluation is done considering as test set the 63 valid entries found with the dataset evaluation. As discussed before we have two different models to find the entities in a text, one uses Babelfy and the other uses the dependency tree and POS tags of the words in the sentence with Spacy and Babelnet. Since we have not the domain information for each question, a single entity recognition task is considered successful if the babelnetID in c1 or in c2 is in the set of the ones predicted by the model. The following are the results on the two different models.

All the data is in *babelfy.txt*, *spacy_pos.txt* and elaborated in *stats.xlsx* in the output folder.

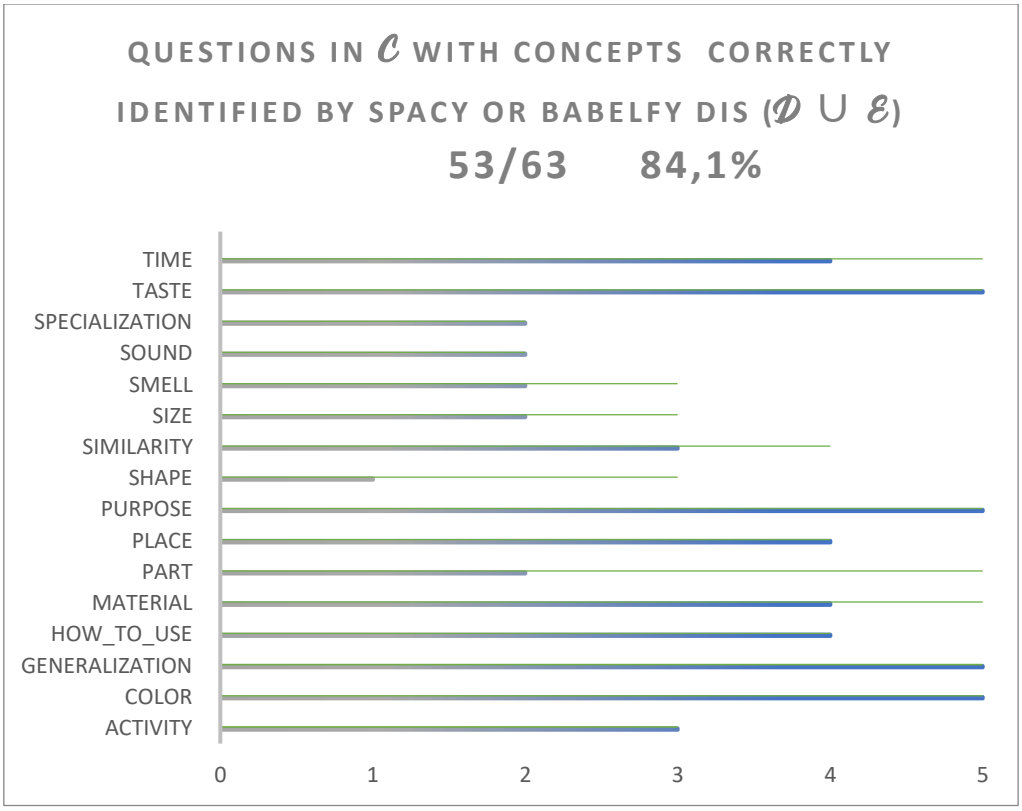




As we see the Babelfy based entity finder has a better performance, on the other hand the Spacy based performs better on some relations, TIME, PLACE and COLOR. This result could be a consequence of the simpler question pattern of those relations, where the deduction of the correct entity from the subtree containing the subject or the object is more straightforward. Comparing the two models we obtain



These results make possible to boost the entity recognition model combining the two approaches. Using the Spacy based entity finder on the TIME, PLACE, PART and COLOR relations and the Babelfy based one on the others we obtain better results.



Relation identifier

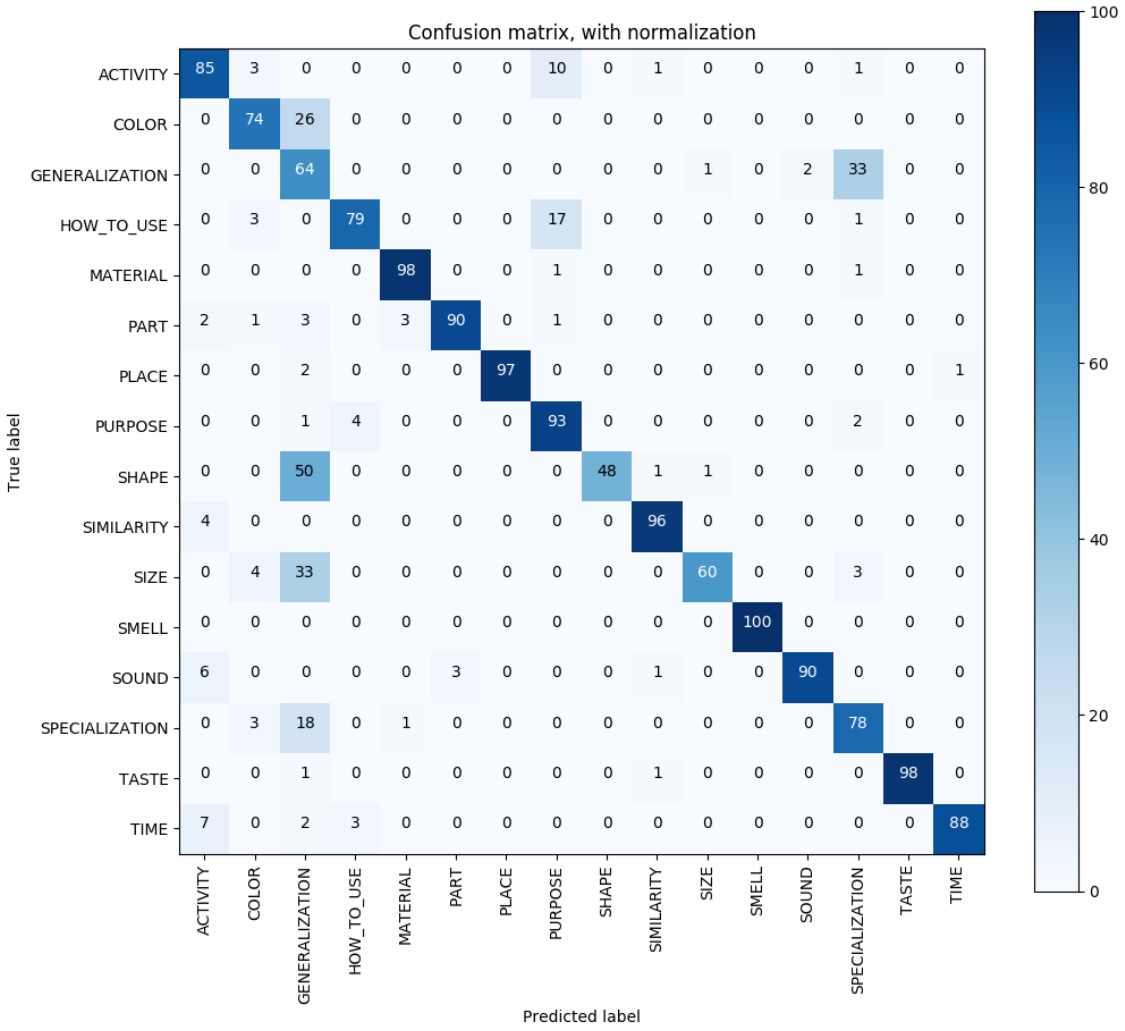
Since the relation identifier is trained only with question patterns, the presence of corrupt entries from the KBS in the test set is less significant in the evaluation of the performance. Almost all the queries that make no sense have a valid question pattern and their relation is tagged properly.

The test set are 1600 couples query-relation, 100 entries for each relation. The following log shows the score for each relation while all the misclassified queries in the format -True Relation, Predicted Relation, Question- are reported in *relation_identifier_performance.txt*.

	precision	recall	f1-score	support
ACTIVITY	0.82	0.85	0.83	100
COLOR	0.59	0.74	0.66	27
GENERALIZATION	0.35	0.64	0.46	100
HOW_TO_USE	0.92	0.79	0.85	100
MATERIAL	0.96	0.98	0.97	100
PART	0.97	0.90	0.93	100
PLACE	1.00	0.97	0.98	100
PURPOSE	0.76	0.93	0.84	100
SHAPE	1.00	0.48	0.65	100
SIMILARITY	0.96	0.96	0.96	100
SIZE	0.97	0.60	0.74	100
SMELL	1.00	1.00	1.00	100
SOUND	0.98	0.90	0.94	100
SPECIALIZATION	0.66	0.78	0.71	100
TASTE	1.00	0.98	0.99	100
TIME	0.99	0.88	0.93	100
avg / total	0.88	0.84	0.85	1527

Notice the little support on the COLOR relation, in the clean dataset there are only 27 entries about colour over more than 700000. The classification score is satisfying almost in all relations except for COLOR, GENERALIZATION and SHAPE.

Before discussing the reasons of those poor performances, it is the case to take a look at the confusion matrix.



The confusion matrix shows interesting patterns:

- The classifier marks as GENERALIZATION several COLOR, SHAPE, and SIZE entries. Looking in detail at the misclassified queries, the principal cause of misclassification is the fact that the word characterizing the relation is the concept itself, absent in the query pattern (e.g. "Can you give me an example of *twisted* thing ?" without twisted it is hard to predict the SHAPE relation).
- The classifier often confuses GENERALIZATION entries with SPECIALIZATION entries and vice versa. This is simply due to the confusion of the humans that compile the database, some entries in the KBS are misclassified *a priori*. (e.g. "What is Cambridge Public School?" is evidently GENERALIZATION but is tagged as SPECIALIZATION).

Comments and Conclusions

The chatbot models and implementation work properly, in particular the entity detection and the relation recognition performance is satisfying (in a well-defined question, entities are identified in the 84% of cases, the f1 score of the relation identifier on all classes is 85%). The speed of the bot is remarkable despite the heavy use of APIs and the searches in the dataset. On the other hand, the quality of the dataset has proved to be poor, less than 80% of entries are well-defined, some makes no sense, some have incorrect relations and entities tags. The wide range of knowledge considered permits the inclusion of very specific entries, that are the major part of the KBS, on the other hand this wide inclusion makes the described entity-relation couples very sparse in the universe of all possible couples, so during a regular user experience most of the time the chatbot answer is "Sorry, I have not an answer for this question". A dataset restricted to certain domains and with a better coverage seems more suitable.

Implementation details

Every src file is well documented, every function has its description explaining the accomplished task and the input and output variable. In the following is reported the general description of every src file (Also present in the code).

get_data
Collects the data from the knowledge base server and store it in a convenient json format in dataset.json. The queries with answer = 'no' are excluded.

babelnet_id_cleaner
Standardizes the data in dataset.json, the clean dataset is stored in clean_dataset.json.

entity_finder
Manages all the interactions with the babelnet APIs, in particular it contains the functions that returns the relevant entities of a given query.

To find the entities are implemented two different strategies:

1. babelfy_disambiguation, finds the entities using Babelfy.
   Faster and cheaper in babelcoins, but as general approach has some patologies (e.g. long questions with several entities)
2. spacy_disambiguation, manually finds the entities considering the POS tags and the dependency grammar tags.
   Slower and more expensive in babelcoins but query-oriented and so without the patologies of babelfy_disambiguation.

If the code is executed standalone, it is provided an example of usage of the functions

Relation_identifier
Finds the type of relation expressed in a given query, this is accomplished using a support vector machine trained on the provided query patterns for each relation. It is used a n-gram representation with unigrams and bigrams.

Concerning the implementation, the "Relation_identifier" class manages the classifier, it prepares the dataset when invoked and has a "training" and "predict" method.

Commented code is used for a grid search to find the best parameters for the classifier.

Answerer
Core of the project, the class Answerer contains the method "answer" that given a couple (query, domain) returns an answer, and contains the method "query" that given a domain returns a query about something not in the dataset.

If the code is executed standalone, it is provided an example of usage of the functions

telegram_chatbot
This is the Telegram bot, it manages the interactions with the users calling the Answerer and the enrich_database function when needed.

The core function is "answer"

enrich_database
Manages the requests of updating of the dataset with the user answers and practically updates the KBS server when executed standalone.

performance_evaluation
Generates the suitable test set and then the evaluation data for the entity finder and the relation identifier.

Easy step-by-step guide

Unpack the zip file wherever you want, go in src and run telegram_chatbot.py. When "setup done" is printed the bot is ready to interact on Telegram.*

The bot name on telegram is RobotSPD13_bot. Press start and begin the interaction.

*If you don't have installed the spaCy library, the bot can still work: set the variable spacy_dis=False in entity_finder and in enrich_database then comment the line `self.spacy_nlp_model = spacy.load('en')` in the initialization of the Answerer class in Answerer.