Question Answering Systems

Jaime Arroyo Morales - LJ1806201
Beihang University

17th January, 2019

Abstract

In computer science, Question Answering (QA) is a field whose main objective is to build systems that are able to understand and answer natural language questions proposed by humans. This systems are gaining a lot of popularity thanks to their usage in search engines and specially in virtual assistants like Siri, Cortana or Alexa. QA discipline is a combination information retrieval techniques and natural language processing and the recent improvement in these two fields and the existance of better knowledge bases have had a lot of impact on QA systems, leading them to better results. In this paper I analyze different approaches to this problem and propose a model to answer simple factoid english questions based on the DBpedia ontology, first I use SVM leaning techniques in order to classify the questions by answer type and then NLP techniques with the objective of processing the question and querying the database. In addition, I propose and algorithm to make the system able of answering spanish questions and achieve in that way cross-language knowledge transfer.

1 Introduction

1.1 Problem definition and Motivation

In the beggining of Question Answering systems the studies had the objective of creating a system able to interact with a human being. This systems have evolved from simple interactions without any database behind to very complex systems able to understand and query questions all over the web and get reliable answers.

This field is different from key word based search engines, whose objective is to reply with a set of documents related to a set of words input as query. QA goes one step further and aims to accept natural language questions as input and crawls through as many documents as possible to find the correct answer. This is possible thanks to the constantly growing internet and also helps to convice users that all the information retrieved is reliable.

Many factors have made this discipline grow and gain more interest recently. Probably the main factor is the users demand to have access to correct information in a efficient and fast way which makes QA a very interesting topic to research on. This demand comes because of the existance of good search engines and the popularity of all the smartphone assistants or even home assistants, which make the task of Question Answering a basic need in a lot of people's lifes.

Other factors like the exponentially increase of availeable information on the internet, as mentioned before, and improvements in natural language processing (NLP) have obviosly had a huge impact on QA systems.

Solving this problem is not an easy task since even with well structured data provided by knowledge bases, the gap that still exists between this and the natural language is very big and one incorrectly parsed question leads to a wrong query that will most likely result in an incorrect answer. While it is true that huge improvements are being made there is still a lot of work and research to do in this field, specially when it comes to answering complex questions, those that involve more than one fact.

1.2 Structure of the work

In this paper I will focus in basic concepts and several approaches to the problem.

In chapter 2 I will talk about different models trying to analyze them and understand the techniques used and will also take a look at the current state of the art in QA discipline.

Chapter 3 will be the explanation of my QA model proposal covered with a lot of details about every step and finally chapter 4 will cover all the aspects related to the cross-language knowledge transfer.

At the end I will analize the results obtained by my model and get some conclusions.

2 Methods

Two of the earliest Question Answering Systems were BASEBALL, a system that answered questions about the US baseball league, and LUNAR, that answered questions about rocks returned by Apollo moon misions. These were restricted-domain systems that relied on a knowledge system hand-written by experts and obtained very good results.

Other Question Answering Systems are those that rely on reading comprehension in order to answer the questions. A very interesting case is the SQuAD dataset, described by the Stanford researchers as a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia

articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage, or the question might be unanswerable. It is very interesting since it has a leaderboard that allows you see the state-of-the-art systems in reading comprehension.

Rank	Model	EM	F1
	Human Performance *Stanford University* (Rajpurkar & Jia et al. '18)	86.831	89.452
1 Jan 10, 2019	BERT + Synthetic Self-Training (ensemble) *Google AI Language* https://github.com/google-research/bert	**84.292**	**86.967**
2 Dec 21, 2018	PAML+BERT (ensemble model) *PINGAN GammaLab*	83.457	86.122
2 Dec 16, 2018	Lunet + Verifier + BERT (ensemble) *Layer 6 AI NLP Team*	83.469	86.043
2 Dec 13, 2018	BERT finetune baseline (ensemble) *Anonymous*	83.536	86.096
3 Dec 15, 2018	Lunet + Verifier + BERT (single model) *Layer 6 AI NLP Team*	82.995	86.035
4 Jan 10, 2019	BERT + Synthetic Self-Training (single model) *Google AI Language* https://github.com/google-research/bert	82.972	85.810
5 Dec 16, 2018	PAML+BERT (single model) *PINGAN GammaLab*	82.577	85.603

Figure 1: Comprehension leaderboard

As we can see in the leaderboard, current state of the art in that field is by BERT, an unsupervised and deeply bidirectional system for pre-training NLP. The system analyzed on this paper, on the other hand, will be different than these two types mentioned before, since it is going to be a open-domain QA system. Open-domain systems are those that aim to return an answer in response to human's questions but are not bounded by any form.

3 The model

In this section I am going to talk about the model I propose and explain with a lot of detail each component. As mentioned before, it is a system able to understand and answer simple english questions and also spanish ones if using the cross-language transfer.

It is very important to describe which questions are supposed to be answered by the system and which are not. Simple questions are those questions that only require to know one fact in order to be answered, for example "What is the capital of Norway?". This will be explained with more detail in the Knowledge-base subsection.

This system is based on the DBpedia ontology, database that will provide the answer to the questions. With DBpedia as our source of information, the main goal is going to be transforming our natural language question into a SPARQL query and parse and validate the answer given by it.

As described by C. Gailer, S. Kohl and S. Oberauer, researchers in IICM, a Question Answering system can be divided in 4 major parts: Question Analysing, Preparing the Dataset, Text Processing and Data Mapping.

First of all, the question needs to be analyzed in order to identify at least the semantic of the answer expected. The next step is to convert our dataset into datastructures understandable by our system. Text processing consists in all the tasks involved in transforming the input question into a SPARQL query and the last step is the execution of it.

These are the 4 subsections I am going to use to describe the model, a variation of the model proposed by A. Tahri and O. Tibermacine with the addition of cross-language transfer and other changes.

3.1 Question Analysing

This is the first step and most important in our architecture since a mistake in this section probably leads to an incorrect answer. The main goal of this step is to be able to classify questions by the type of answer, crucial to validate the answer provided by our system. A simple example could be: given the question 'When did Michael Jackson die?' we expect a number or a date of an answer. If given any other kind of answer we can discard that answer and look for it again.

While there are some approaches to tackle this question classification problem, the best performances have been achieved by machine learning, especially Support Vector Machines. With the technique decided, another important thing to take into account is in how many classes you want to classify the questions. I will adopt the 6 classes proposed by Li and Roth in their research, same researchers that have proved with a comparative study that SVM is currently the state of the art technique in Question Classification.

The 6 classes are the following:

- Human (HUM)
- Location (LOC)
- Entity (ENTY)
- Description (DESC)
- Numeric (NUM)
- Abbreviation (ABBR)

Using the same example as before, when input the question 'When did Michael Jackson die?', the system should answer NUM, since a year is expected.

First of all, I am going to explain the basics of SVM algorithms, which will be very usefull to be able to understand properly the implementation of Question Classification used by my model.

3.1.1 Support Vector Machines (SVM)

Support Vector Machine is a highly used machine learning algorithm whose objective is to find a hyperplane in a N-dimensional space, where N is the number of features used to analyze data. This hyperplane will be used to classify our data into the different classes. SVM produces an acceptable accuracy with less computation power than other Machine Learning algorithms and while it can be used also for linear and logistic regression problems, his main usage is multi class classification.

Hyperplanes are the boundaries we are going to use to calssify the data. Our data will be transformed into a numerical vector which will create a point in the space in a process called feature extraction since most of the times our data will not be already provided as a numerical vector. Feature extractio is a topic that will be covered later in this paper. Once we have set a point in the space it is trivial to determine the class it belongs since we only have to chech in which side of the hyperplane that point is falling.

The other important element is this algorithm are the support vectors, which are those points that are closer to the hyper plane and can be used to change the position and orientation of the planes in order to maximize the margin of the class. These points will be key in order to build our SVM. In figure 2 we can see two examples of a 2D and 3D hyperplane that clearly acts as a boundary between classes and figure 3 helps us visualize the utility of support vectors and having a higher margin.

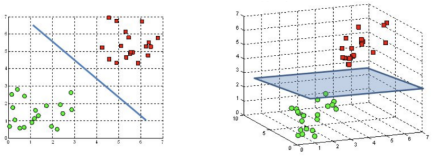

Figure 2: 2D and 3D hyperplanes

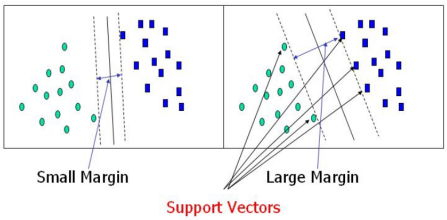

Figure 3: Support Vectors

As seen before, SVM are linear functions $f(x) = w \cdot x + b$ where $w \cdot v$ is the inner product between the weight vector and the input vector(features vector) and b is the bias, a value that will allow us move the

hyperplane in parallel directions to itself. W (weight vector) is a series of values that will determine the orientation of the hyperplane and for that reason these are the values that our system will have to learn. It is a bynary classifier where we take the output of the linear function and identify the input vector as one class if the output is grater than 1.

This algorithm is based on statistical learning and it is a supervised learning model, which means that to train it we will need already labeled data to work with. SVM will analyze the training dataset and will produce as output the linear function mentioned in the previous paragraph. This consists into being able to generalize the training dataset to unseen situations.

Like in all the supervised learning algorithms we need to define a loss function, a function that will compare our predicted value with the correct value of a certain input. What we need to obtain here is the highest margin between points and the hyperplane possible and the loss funtion used is the hinge loss function with an added parameter used to balance the margin maximization and the loss.

$$min_w \lambda \parallel w \parallel^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)_+$$

Figure 4: Loss function for SVM

Having the loss function, we have to take the partial derivates with respoect to the weights in each iteration to find the gradients and use them to update out weights vector. When no misclassification is made, we only have to update the weights but if mistakes have been made we will have to include the loss with the regularization parameter to perform the update.

3.1.2 Question Classification Implementation

Lately multiple machine learning frameworks are being developed and they offer you libraries with the basic Machine Learning algorithms which make creating machine learning based systems much easier since no full knowledge of these algorithms is required in order to use them. The full implementation of these algorithms is not in the scope of this work. This whole project is implemented using Python and a good existing framework is *scikit-learn*, an Open Source package bult on NumPy, SciPy and matplotlib that offers the user simple and efficient tools for data mining.

For multi-class classification problem there are two main approach strategies: *one-vs-one* and *one-vs-the-rest*. The main difference between these two is the number of classifiers that are going to be constructed. Being N the number of classes, the first will need N·(N-1)/2 classifiers while the one-vs-the-rest strategy will only need N. The one-vs-one classifier is less sensitive to imbalaced datasets but it is more computationally expensive.

The first decission has to be taken here and I decided that *on-vs-the-rest* strategy would be more suitable to this problem. The reason for that is that since we are working with word sentences, the feature extrator used in the next steps will probably generate a high number of numeric features and since the number of classifiers needed by the one-vs-one strategy is quadratic depending on the number of features, the cost of computing all these procees is much higher and does not compensate with providing higher accuracy.

In figure 5 we can see the basic function calls to build a SVM with scikit-learn package. The class used is LinearSVC which is the one that implements the one-vs-the rest strategy and after tuning the parameters I realized that the default ones

```
>>> lin_clf = svm.LinearSVC()
>>> lin_clf.fit(X, Y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> dec = lin_clf.decision_function([[1]])
>>> dec.shape[1]
4
```

Figure 5: Linear SVC implementation

were already giving me results with an acceptable accuracy for this project so those are the ones I am going to be using.

Our objective now is to find a good dataset and transform it into the numerical vectors that we are going to give to the *fit* function as the X parameter. Y parameter is going to be the list of labels.

3.2 Preparing the dataset

Now that we know the input requirements for our system we need to look for a dataset that suits our needs and transform it into the data types accepted by our system. The two main requirements are that we need already labeled data, since we are working with a supervised learning algorithm, and a vector of features for each sample in our dataset.

The chosen dataset is TREC10 dataset. This provides 5500 already labeled questions for the training of the model and also a set of test questions to try our system in never seen before questions. These questions are provided in the form of *class:subclass question*, for example 'HUM:ind Who invented Make-up ?'. HUM means that we are expecting a human as the answer to the question and *ind* is the subclass, which I am not going to use in this project. We will need simple regular expresion algorithms to extract the information and storing it in two diferent vectors, one for the questions and one for the labels. In the code snippet of figure 6 we can see the RE algorithm.

At this point we have to focus in the *questions* vector since as we mentioned before, need to transform it into a numberic features vector. Again the *scikit-learn* library will help us with the process. The first utility provided is the tokenization, a process that will transform our questions into a bag of words that will be used to givean integuer id for each token. The next step is counting the occurences of tokens in each sentence, crucial step that will help us weight and evaluate the importance of eache word. The main perk of using this strategy is that it allows us ignore the relative position of each word in a sentece. This process of extracting features by transforming string questions into numerical features ir called vectorization.

The result of this process will be a very huge matrix of size [number of questions, number of features], where number of features will be the number of different tokens found in the dataset document. This matrix will contain many values that are 0 as soon as the dataset contains a couple of sentences with different words. Thankfully this Python package provides funtion in order to speed up operations with this matrix and memory usage.

The Python class that implements the tokenization and counting is called *CountVectorizer* and again the default values seem to work good with the TREC10 dataset. In figure 7 we can see the default parameters.

In later steps of the system we will have to process the dataset again in order to ex-

```
def processInput(path):
        labels = []
        questions = []
        with open(path, 'r') as f:
                RE = re.compile('([A-Z]*):([a-z]*)\s(.*)')
                for line in f:
                        reg = RE.search(line)
                        labels.append(reg.group(1))
                        questions.append(reg.group(3))
        return [labels, questions]
```

Figure 6: Regular expression snippet

```
>>> vectorizer = CountVectorizer()
>>> vectorizer
CountVectorizer(analyzer=...'word', binary=False, decode_error=...'strict',
        dtype=<... 'numpy.int64'>, encoding=...'utf-8', input=...'content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern=...'(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

Figure 7: CountVector class default parameters

tract keywords and resources but in a completely different process, for now transforming both training and test dataset into their corresponding matrix is all we need to call the previously mentioned scikit-learn function *fit*.

Once acceptable accuracy is obtained we have to serialize the LinearSVC object. Serialization is the process of converting a programming object into a binary format that can be stored. This will allow us load the SVM into a program when we need it instead of having to do all the preprocessing and training again. In Python this is provided by the *Pickle* package, which contains the function *dump* that as the name tells us, allows duming an object into a *bin file*.

3.3 Text Processing

Now that we already have our question classificator trained, our next step before executing the query is to understand the set of questions that are going to be input to the system, which is not the same set used to train the classificator. By understanding I mean knowing which elements or words of the question are going to be usefull for us to answer it. This will depend on our architecture behind the system, which is the DBpedia ontology, so a brief introduction about knowledge-bases and how DBpedia works is needed to understand further steps.

3.3.1 Knowledge-bases

Knowledge bases are playing an increasingly important role in enhancing the intelligence of Web and enterprise search and in supporting information integration. Knowledge-bases are huge data structures used to store structured and unstructured information. They have a much higher level of abstraction compared to simple databases in order to store the information in a closer way than human brain does. The reason for that is that these bases are often used as artificial intelligence tools by systems that require reasoning. KB are often together with an inference engine, a tool that will help the base infere new facts from the existing ones using logic and other complex algorithms.

Often they are represented as ontologies. D. Man described ontologies as a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. They

are used to describe and reason about properties of a domain since they provide a shared vocabulary that can be used to model so, for example by describing the type of the objects, their properties and relations.

Another important concept to have in mind are knowledge-graphs, a datastructure based on graph theory where nodes are entities or concepts and edges are relations between them. It is very important to know that knowledge-graphs(KG) and knowledge bases (KB) are not the same concept, all knowledge-graphs are knowledge bases but not viceversa, since knowledge bases con be implemented in different ways. The main advantage of using knowledge-graphs is that it is one of the most flexible formal data structures and most important, it is self-descriptive, all the information is stored in natural language so it is easy to query and understand. In figure 8 we can see an example of knowledge-graph.

For this project, the knowledge-base chosen has been DBpedia.

3.3.2 DBpedia

DBpedia is a project started by researchers in Free University of Berlin and Leipzig University that aims to extract structured content from information found in Wikipedia. Wikipedia is the largest enciclopedia in the world and the 5th most visited website in 2018 according to Alexa, it contains over 46 million articles in multiple languages and it is currently growing at a rate of 20,000 new articles per month. These numbers make Wikipedia one of the best sources of information availeable and being able to convert this articles information into structured data allow computer programs to take advantage of all this information.

DBpedia allows users to semantically query relationships and properties of Wikipedia resources, returning RDF triples which can be parsed to get the usefull information of it. The triplet concept is very important since it is the way data is stored into DBpedia ontology. A fact is a triplet <o1, r, o2 >where o1 and o2 are objects/entities and r is the relationship between them, for example ¡Oslo, capital, Norway¿. Other triplets of form <o1, typeof, t >take part in the DBpedia ontology definition and allow us define the type each object in the graph. As mentioned before, our system will only be able to answer questions involving one fact, this means that while it can answer 'Who is the president of the United States of America?', it will not be able to answer 'Who is the wife of the president of the United States of America?' since it involves two facts, knowing who is the president of USA and later knowing who is his wife.

The English version of the DBpedia knowledge base describes 4.58 million things, out of which 4.22 million are classified in a consistent ontology, including 1,445,000 persons, 735,000 places (including 478,000 populated places), 411,000 creative works (including 123,000 music albums, 87,000 films and 19,000 video games), 241,000 organizations (including 58,000 companies and 49,000 educational institutions), 251,000 species and 6,000 diseases.

The main question now is how to query DBpedia Ontology. Its data is served as Linked data and can be navigated throught complex queries with SQL-like languages. For this project the selected query language is SPARQL .

3.3.3 SPARQL

SPARQL is a query language used to retrieve and manipulate data stored as Resource Description Framework(RDF) format. As a query language, SPARQL is "data-oriented" in that it only queries the information held in the models; there is no inference in the query language itself.

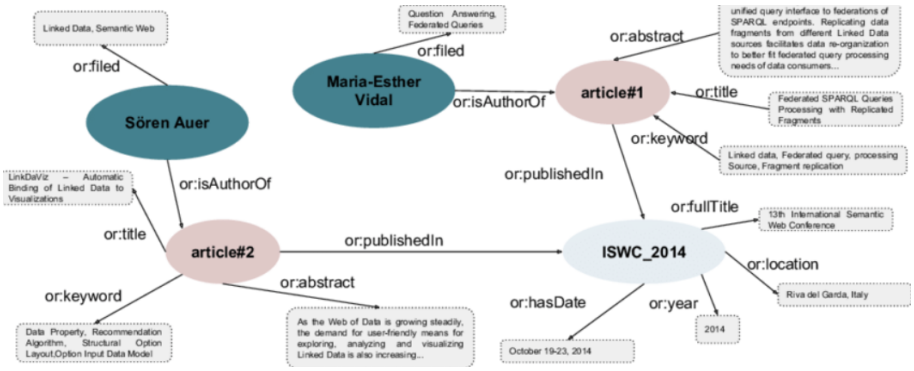

Figure 8: Knowledge-graph example

Queries have two main parts, the Solution Modifier, that provides the basis for categorizing different types of query solutions, and the Query Body, which contains a serie of statements that represent the entities relationships. This statements are represented using Turtle Notation, for example, *<spiderman ><relationship/enemyOf <green-goblin >*.

In figure 8 we can see an example of a simple query that looks for names of people that are surnamed *Smith*. It is very important to learn the basic structure of the different query types since we will have to create a program able to generate them automatically. The main Solution Modifiers used in this project are SELECT, which acts the same way than a normal SQL Select, and ASK, where we will write a set of triplets and the system will return Yes or No depending if these triplets exist on the DBpedia ontology. ASK will be very usefull to validate the type of the answer.

3.3.4 Resource extraction with DBpedia Spotlight

Before moving ¡nto the implementation, another useful tool used in this project has been DBpedia Spotlight. When trying to answer the questions we need to detect which entities are on it and to which entity does the property we are asking belong.

DBpedia spotlight is the perfect solution to this problem, it is a tool designed to detect DBpedia resources in text. It performs 4 steps to do so, the first one is *Spotting*, which consists in the identification of surface forms substrings of the original input that may be entity mentions. after that it selects the set of the surface forms identified that are candidates. The third step is the disambiguation, deciding on the most likely candidate resource for each selected surface form. The last step is filtering, adjusting the annotations to task-specific requirements according to user-provided configuration.

We can access to this tool via its API. There are multiple ways of making calls to its API, I use the linux command *curl* to get the resources information. A possible command could be *curl "$http://api.dbpedia-spotlight.org/en/annotate ?text=$question &confidence=0.8&support=20" -H "Accept:text/xml.*

In this query we can see that there are two parameters, confidence and support. Confidence is the Disambiguation Confidence parameter, it is a threshold which takes a value between 0 and 1. Setting a high value will give us better and reliable solutions but we may be risking losing other correct resources too. The other paramenter is Support, the Resource Prominence parameter, it helps us ignore unimportant or uninformative resources.

The result of this command will be an

xml file containing the list of resources detected and more usefull information. We can see an example with the question *How old is Michael Jackson?* in figure 9. There are different fields we will need out of it. The most important field is *similarityScore*, this is a number between 0 and 1 and we will select the resource with the highest since it is the one that DBpedia is more sure about. Another important field is the URI, which actually is what we were looking for. That URI is the link to the DBpedia resource that we are going to query later. The last attribute we are going to use from this XML is the *surfaceForm* field, which indicates the words Spotlight used to identify the resource. We need them to discard them later on the keyword selection.

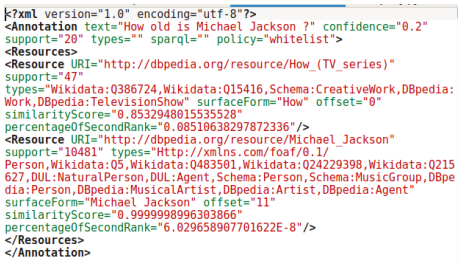

Figure 9: Resource's XML

3.3.5 Obtaining resource's properties

Now that we have identified the main resource in the question, we will need to list all its availeable DBpedia properties so we can later decide which is the one we are interested in. At this point we will have to execute our first DBpedia query, which will come as shown here:

```
SELECT ?Property
WHERE { <Resource> ?Property ?h .  }
GROUP BY ?Property
```

Being *Resource* the URI of the resource obtained by DBpedia Spotlight. In Python, the best solution to query DBpedia is by using the SPARQLWrapper class and creating a SPARQLWrapper instance around *http://dbpedia.org/sparql*. The query is input as a String and the result will be given in the chosen format. In my program the result is obtained as JSON, since it makes it very easy navigate through it and extract the information but I also find interesting to look at the HTML result, since it provides a clear idea of what we are queriing. In the next figure we can see some of the results obtained queriing for Michael Jackson's properties.

Property
http://www.w3.org/2000/01/rdf-schema#label
http://www.w3.org/2000/01/rdf-schema#comment
http://dbpedia.org/ontology/deathDate
http://dbpedia.org/property/birthPlace
http://purl.org/dc/terms/subject
http://xmlns.com/foaf/0.1/surname
http://dbpedia.org/property/restingPlace
http://www.w3.org/2000/01/rdf-schema#seeAlso
http://www.w3.org/2002/07/owl#sameAs
http://dbpedia.org/property/birthDate
http://dbpedia.org/property/title
http://dbpedia.org/ontology/abstract
http://dbpedia.org/ontology/background
http://dbpedia.org/ontology/recordLabel
http://dbpedia.org/property/deathDate
http://purl.org/linguistics/gold/hypernym
http://xmlns.com/foaf/0.1/gender
http://dbpedia.org/ontology/associatedBand
http://dbpedia.org/property/caption
http://dbpedia.org/property/wordnet_type
http://dbpedia.org/ontology/activeYearsEndYear
http://dbpedia.org/ontology/genre
http://dbpedia.org/property/netWorth
http://dbpedia.org/property/parents
http://dbpedia.org/ontology/deathPlace
http://dbpedia.org/ontology/birthPlace
http://dbpedia.org/ontology/birthDate

Figure 10: Properties query example

Our next objective will be deciding which of this properties was the one asked in the question.

3.3.6 Keyword analysis

At this point we have already obtained the resources and the list of properties DBpedia has. Our objective now will be to decide which of this properties contains the

solution to the question. It may be the case that the information asked is not in the resource's properties, in that case, the system will not be able to answer since it doesn't have that info.

To do so, we will have to extract words in the question that may be usefull for us. The first step will be convert the question into a bag of words, since from now on the position words had in the sentence is not useful anymore. This can be done with the *nlkt* function *word_tokenize*, which basically splits the question by spaces and creates a list where every position is a word of the question.

Our next step will be removing stop words. Stop words are those than don't add any meaning to a sentence, usually connectors. The list of stop words used for this project is the one provided again by the *nlkt*. We will also remove from the keyword candidate list the resource since if some words have already been tagged as resource, we cannot considere them properties. We can know which words were used as resources thanks to the attribute *SurfaceForm* provided by DBpedia Spotlight. In Python it is very easy to filter words in a sentece and we can do it with just one instruction: *filtered_words = [word for word in text if word not in stopwords.words('english') and word not in surface].*

The last step is running a part-of-speech tagger to the remaining candidate words. The Stanford Natural Language Processing group described POS taggers ass pieces of software that read text in some language and assign parts of speech to each word (and other token), such as noun, verb, adjective, etc. This will be very usefull for us to obtain the final keywords. We will discard all the words that are not tagged as nouns, verbs, adjectives or adverbs and the ones remaining after this process will be considered keywords. The implementation of this is with the pos_tag function provided by the *nltk* package and using the tagset 'universal', since the default one goes into too much detail and we only need to know the basic type of each word.

At this point we have completed the objective of text processing, which was to extract usefull information out of the question. Our next step will be preparing and executing the query.

3.4 Data mapping

This is the last step of the system. The objective here is find which of the resource's properties is the one asked in the question. To do so, we will compare the list of keywords and the list of properties and decide which are more similar.

A double loop will be executed since we will compare all the properties against all the keywords, being the outer loop the one that iterates through the properties. Working in this way will allow us store a final score for each property and finally decide our answer.

First of all, it is important to mention that the comparison will be done not only against keywords, it will also be compared to synonims or related words of keywords in order to be more precise. The tool used to extract synonims is the API to the Merriam-Webster dictionary, America's most trusted online dictionary for English word definitions, meanings, and pronunciation. In order to use it, it is required to generate an API key registering in the website but the usage is completely free.

In Python we can query the dictionary and obtain a XML object using the *urllib* library and queriing to the url *https://www.dictionaryapi.com/api/v1/references/thesaurus/xml/X?key=Y*, where X is the word we are queriing and Y our generated API key. In figure 11 we can see the example found on the Merriam-Webster API website, which shows que XML output for the query of the word

umpire. From this XML we will extract the words obtained in nodes *syn*, which corresponds to synonims and *rel*, which corresponds to related words.

Now we will finally do the comparison without the list of words and the list of properties. There are multiple algortihms that take two words as input and return a value depending on how similar they are. Three different algorithms were tested on this project.

The first algorithm used was the Levenshtein distance, also known as the minimum edit distance algorithm. This algorithm consists in, given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2':

- Insert
- Remove
- Replace

For example, the edit distance between *cat* and *cut* would be 1, since we can replace letter a by u and we already have the objective word.

The second distance measure used was the Jaccard similarity, defined as the size of the intersection divided by the size of the union of two sets. We can see the formula and the python implementation in the following two figures.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Figure 12: Jaccard similarity formula

```
def DistJaccard(str1, str2):
    str1 = set(str1.split())
    str2 = set(str2.split())
    return float(len(str1 & str2)) / len(str1 | str2)
```

Figure 13: Jaccard similarity Python implementation

Althought this algorithm was proved to achieve better results than the minimum edit distance algorithm, one last algorithm was tested, the cosine similarity.

Cosine similarity is the cosine of the angle between two n-dimensional vectors in an n-dimensional space. It is the dot product of the two vectors divided by the product of the two vectors' lengths (or magnitudes). Again I will show the mathematical formula and my implementation in Python.

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Figure 14: Cosine similarity formula

```
def word2vec(word):
    from collections import Counter
    from math import sqrt
    cw = Counter(word)
    sw = set(cw)
    lw = sqrt(sum(c*c for c in cw.values()))
    return cw, sw, lw

def cosdis(v1, v2):
    v1 = word2vec(v1)
    v2 = word2vec(v2)
    common = v1[1].intersection(v2[1])
    return sum(v1[0][ch]*v2[0][ch] for ch in common)/v1[2]/v2[2]

def distance_words(w1,w2):
    return 1 - cosdis(w1,w2)
    #return nltk.edit_distance(w1,w2)
```

Figure 15: Cosine similarity Python

Both Jaccard and Cosine similarity were giving acceptable results but I finaly decided to use the Cosine Similarity as distance mesurement. The next step is to compute these similarities and sort the properties vector by similarities.

Here comes our last step in the process. We take the property that we have found more similar to any of the keywords and we query it to DBpedia. The query is very simple, we are asking the database for a triplet where we already know 2 values, the resource and the property, the sample query would be *SELECT ?result WHERE* { *<Resource><Property >?result* }. DBpedia endpoint will search through and complete the triplet, returning the answer to the variable result.

We will query for the most similar

```
<entry id="umpire">
  <term>
    <hw>umpire</hw>
  </term>
  <fl>noun</fl>
  <sens>
    <mc>
      a person who impartially decides or resolves a dispute or controversy
    </mc>
    <vi>
      usually acts as
      <it>umpire</it>
      in the all-too-frequent squabbles between the two other roommates
    </vi>
    <syn>
      adjudicator, arbiter, arbitrator, referee, umpire
    </syn>
    <rel>
      jurist, justice, magistrate; intermediary, intermediate, mediator, mediatrix
    </rel>
  </sens>
</entry>
```

Figure 11: Merriam-Webster query example

property first and after getting the result our last step will be validate that the answer to that question is compatible with the one our question classification predicted. Depending on the type of answer predicted by the classificator, different procedures will be performed. The two most interesting cases are the human(HUM) and location(LOC) tags. It is very important to know that the answer given by the DBpedia can be a value or the URI of another resource. For human predicted questions, we will have to do one last query to DBpedia to guarantee that the solution is correct, or at least, compatible with the type of answer. We will considere valid a Human answer when the result obtained by DBpedia is an URI with that contains the type Person. At this point is when we are going to use the ASK Solution Modifier. ASK expects a series of triplets and returns if those triplets exist in DBpedia, by doing the query *ASK WHERE* { *<ANSWER><http://www.w3.org/1999/02/22-rdf-syntax-ns#type><http://dbpedia.org/ontology/Person >.* } , being *ANSWER* the URI obtained as answer to the question, we will accept the answer if the result of this query is true, otherwise we will discard it. The same procedure is done with the location questions, where we query for *http://dbpedia.org/ ontology/Location* instead. All the other types have basic comprovations, like ENTITY expects an URI as answer, NUMBER expects some number or date on it, and finally the other types, that expect text as an answer.

If the answer is accepted is given as final answer and if it is not, all these comprovations will be made again with the second property more similar calculated before. This will be done until a valid answer is found.

4 Cross language information transfer

As mentioned before, Wikipedia is the biggest enciclopedia in the world, containing more than *5,700,000* articles in En-

glish. But on the other hand, in Spanish it contains 1,5 milion articles, which is a lot, but still 4 times less articles than in English, which means that it contains aproximately 4 times less information in the DBpedia. Things get worse it other language. With some small changes, the model proposed before can answer Spanish questions using the English DBpedia.

Translating the whole question can lead to many errors or can be computationally expensive, but if keyword identification can be performed before, only the translation of keywords would be needed, which is easier to compute since no context is needed.

Since no good enough Spanish Part-of-Speech Taggers could be found, the stop words removing process becomes more important. First, stop words are removed from the bag of words formed by the question. The list can be found in *https://www.ranks.nl/ stopwords/spanish*, a website that also provides stop words list for other languages. The spanish list contains 179 words.

After this filtering, another filter will be passed in order to remove Question Pronouns from the sentence, being those *Why, What, Which, When, Where, Who and How*, in Spanish *Por qué, Qué, Cual, Cuando, Donde, Quién and Cómo*. All the remaining words will be considered keywords of the question. These keywords are translated using *Translator* python package as we can see in figure 16. After that, DBpedia Soptlight is run and al the remaining words are used for the properties comparison. Then the same model as before is used and finally the answer is translated back to Spanish.

```
>>> from translate import Translator
>>> translator = Translator(from_lang="es", to_lang="en")
>>> translation = translator.translate("hola")
>>> print(translation)
hello
```

Figure 16: Python translation snippet

5 Conclusion

In this paper I reviewed some state of the art QA systems and proposed mine, with the first part involving machine learning techniques and the second involving NLP techinques. The system achieves a 85% of accuracy classifying questions on the TREC10 dataset which is a very acceptable result. The question answering was tested against a manually created dataset where we can conclude that the system achieves good performance in simple questions in which the information in DBpedia is stored the same way that the structure of the question. The system has more problems with questions where a further understanding of the keywords is needed or the list of synonims doesn't match the property name we are looking for. Future work in this project will involve working on these aspects in order to improve the system's accuracy.

References

[1] *Baseball: an automatic question-answerer.*, GREEN JR, Bert F; et al. (1961), Western joint IRE-AIEE-ACM computer conference: 219–224..

[2] *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, (Submitted on 11 Oct 2018)

[3] *Question Answering Over Knowledge Graphs: Question Understanding Via Template Decomposition* Weiguo Zheng, Jeffrey Xu Yu, Lei Zou, Hong Cheng, The Chinese University of Hong Kong, China; Peking University, China.

[4] *New Trends in Automatic Question Answering* Christian Gailer, Stefan Kohl, Stephan Oberauer

[5] *Support vector machines*, Marti A. Hearst, University of California, Berkeley

[6] *DBpedia and the live extraction of structured data from Wikipedia* Mohamed Morsey, Jens Lehmann, Sören Auer, Claus Stadler, Sebastian Hellmann, (2012)

[7] *DBpedia based Factoid Question Answering*, Adel tahri and Okba Tibermacine

[8] *ONTOLOGIES IN COMPUTER SCIENCE*, Diana Man, DIDACTICA MATHEMATICA, Vol. 31(2013), No 1, pp. 43-46

[9] *Scikit-learn: Machine Learning in Python*, Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., Journal of Machine Learning Research, 2011

[10] *Natural Language Processing with Python.* Bird, Steven, Edward Loper and Ewan Klein (2009).