# PiQASso: Pisa Question Answering System

Giuseppe Attardi, Antonio Cisternino, Francesco Formica,
Maria Simi, Alessandro Tommasi
*Dipartimento di Informatica, Università di Pisa, Italy*
{attardi, cisterni, formicaf, simi, tommasi}@di.unipi.it

*"Computers are useless: they can only give answers" – Pablo Picasso –*

**Abstract**

PiQASso is a Question Answering system based on a combination of modern IR techniques and a series of semantic filters for selecting paragraphs containing a justifiable answer. Semantic filtering is based on several NLP tools, including a dependency-based parser, a POS tagger, a NE tagger and a lexical database. Semantic analysis of questions is performed in order to extract keywords used in retrieval queries and to detect the expected answer type. Semantic analysis of retrieved paragraphs includes checking the presence of entities of the expected answer type and extracting logical relations between words. A paragraph is considered to justify an answer if similar relations are present in the question. When no answer passes the filters, the process is repeated applying further levels of query expansions in order to increase recall. We discuss results and limitations of the current implementation.

## 1. Architecture

The overall architecture of PiQASso is shown in Figure 1 and consists in two major components: a paragraph indexing and retrieval subsystem and a question answering subsystem.

The whole document collection is stored in the paragraph search engine, through which single paragraphs are retrieved, likely to contain an answer to a question.

Processing a question involves the following steps:

- question analysis
- query formulation and paragraph search
- answer type filter
- relation matching filter
- popularity ranking
- query expansion.

Question analysis involves parsing the question, identifying its expected answer type and extracting relevant keywords to perform paragraph retrieval. The initial query built with such keywords is targeted to high precision and to retrieve a small number of sentences to be evaluated as candidate answers through a series of filters. This approach was inspired by the architecture of the system FALCON [5]. PiQASso analyzes questions and answer paragraphs by means of a natural language dependency parser, Minipar [2].

The semantic type filter checks whether the candidate answers contain entities of the expected answer type and discards those that do not.

A semantic filter identifies relations in the question, and looks for similar relations within candidate answers. Relations are determined from the dependency tree provided by Minipar. A matching distance between the question and the answer is computed. Sentences whose matching distance is above a certain threshold are discarded. The remaining sentences are given a score that takes into account the frequency of occurrence among all answers. The highest ranking answers are returned.

If no sentence passes all filters, query expansion is performed to increase paragraph recall. The whole process is iterated using up to five levels of progressively wider expansions.

PiQASso is a completely vertical system, made by linking several libraries into a single process, which performs textual analysis, keyword search and the semantic filtering. Only document indexing is performed offline by a separate program.

## 2. Paragraph Search Engine

PiQASso document indexing and retrieval subsystem is based on IXE [1], a high-performance C++ class library for building full-text search engines. Using the IXE library, we built a paragraph search engine, which stores the full documents in compressed form and retrieves single paragraphs. However, we do not simply index paragraphs instead of documents: this approach is not suitable for question answering since relevant terms may not all appear within a paragraph, but some may be present in nearby sentences.

Our solution is to index full documents and to add sentence boundary information to the index, i.e. for each document, the offset to the start of each sentence. A sentence splitting tool is applied to each document before indexing.

The queries used in PiQASso consist of a proximity query involving the most important terms in the question

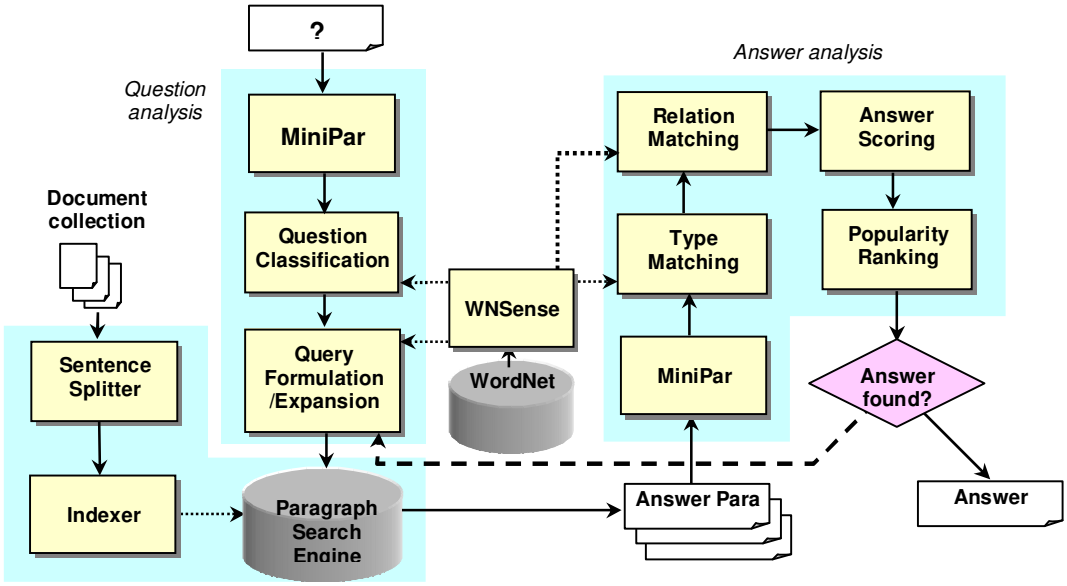


**Figure 1. PiQASso Architecture.**

combined in AND with the remaining terms. Such queries select documents containing both relevant context for the question and paragraphs where the required words occur. The paragraph engine ranks each paragraph individually and extracts them from the source document exploiting sentence boundary information.

The sentence splitter is based on a maximum entropy learning algorithm as described in [9].

Since sentence splitting is quite time consuming, performing it at indexing time improves significantly PiQASso performance. On a 1 MHz Pentium 3, a paragraph search on the whole Tipster collection takes less than 50 msec. Since documents are stored in compressed form, accessing the individual paragraphs requires an additional amount of time for performing text decompression, which depends on the number of results.

## 3. Text analysis tools

Our approach to Question Answering relies on Natural Language Processing tools whose quality and accuracy are critical and influence the overall architecture of the system. We performed experiments with various tools and we had to adapt or to extend some of them for achieving our aims.

We briefly sketch the main NLP tools deployed in PiQASso:

- the dependency parser Minipar
- WNSense: an interface to WordNet
- a Named Entity tagger.

### 3.1. Minipar

Sentences are parsed by means of Minipar [2], producing a dependency tree which represents the dependency relations between words in the sentence. A dependency relationship is an asymmetric binary relationship between a word called *head*, and another word called *modifier*. A word in the sentence may have several modifiers, but each word may modify at most one word. Figure 2 shows an example of a dependency tree for the sentence *John found a solution to the problem*. The links in the diagram represent dependency relationships. The direction of a link is from the head to the modifier in the relationship. Labels associated with the links represent types of dependency relations. Table 1 lists some of the dependency relations.

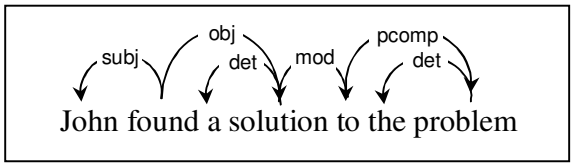


**Figure 2. Sample dependency tree.**

The root node does not modify any word, and is given an empty node type. Other empty nodes may be present in the tree. For instance, in the parse tree for sentence "*It's the early bird that gets the worm*", the word "*that*" is identified as the subject of the "*gets the worm*" subordinate phrase, and an empty node is inserted to

represent the subject of the verb "*gets*", including a reference to the word "*bird*". The parser identifies "*that*" as the subject of "*gets*" and "*the early bird*" as an additional one.

This is an instance of the problem of *coreference resolution*: identifying the entity to which a pronoun refers. From the dependency tree built by Minipar we gather that "*the early bird*" is the subject of "*gets the worm*", enabling us to answer a question like "*who gets the worm*?", even if question and answer are stated in slightly different syntactic forms.

Minipar has some drawbacks: its parsing accuracy is not particularly high and the selection of dependency relations is somewhat arbitrary, so that two similar phrases may have quite different, although correct, parses. We apply several heuristic rules to normalize the dependency tree and to facilitate identifying the most essential and relevant relations for comparing questions and answers.

| Relation | Description |
|---|---|
| i | main verb |
| subj, s | subject of the verb |
| obj, obj*n* | object of the verb |
| pcomp-n | prepositional complement |
| appo | appositive noun |
| gen | genitive |
| inside | location specifier |
| nn | nominal compound |
| lex-mod | lexical modifier |
| det | determiners |
| mod | Modifiers (adjs, advs, preps) |
| pred | Predicate |
| aux | Auxiliary verb |
| neg | negative particle |

**Table 1:** some relations in Minipar output.

Minipar is also capable of identifying and classifying named entities (NE). Using its own internal dictionary, plus a few rules, it detects word sequences referring to a person, a geographic location or an amount of money.

## 3.2. WNSense

We built WNSense (WordNetSense) as a tool for classifying word senses, assigning a semantic type to a word, and evaluating semantic distance between words based on hyperonymy and synonymy relations. WNSense exploits information from WordNet [7], for instance to compute the probability of a word sense.

### 3.2.1. Sense Probability

Senses are organized in WordNet in distinct taxonomies (for instance, the word "crane" has senses in the "animal" taxonomy as well as in the "artifact" one). During sentence analysis PiQASso often needs to determine whether a word belongs to a certain category: e.g. it is of the expected answer type. This can be estimated by computing the probability for the word sense to belong to a WordNet category (e.g., the probability of the sense of the word "cat" to fall within the "animal" category).

WordNet orders word senses by frequency. Given such ordered list of senses $\{s_0, \ldots, s_n\}$ for a word $w$, we compute the probability that the sense for the word belongs to category $C$ as follows:

$$P(w, C) = k \sum_{j=0}^{n} \frac{n-j}{n} \gamma(s_j)$$

where

$$\gamma(s_j) = \begin{cases} 1 & if\ s_j \in C \\ 0 & otherwise \end{cases}$$

and $k$ is a parameter of the heuristics, roughly the probability that the first WordNet sense is the correct one (currently, $k$ is at 0.7).

### 3.2.2. Word Type

The type for a word $w$ is computed as:

$$\arg\max_{C \in TLC} P(w, C)$$

i.e. the category $C$ among those in *TLC*, to which the word belongs with the highest the probability. The *TLC* categories used by PiQASso are the 23 top-level categories from WordNet corresponding to nouns, from the total of 45 lexical files into which WordNet organizes synsets.

### 3.2.3. Word Distance

A measure of word distance is used for estimating the distance between two sentences, in particular an answer paragraph and a question.

Word distance for hyperonyms is based on the distance in depths of their senses in the WordNet taxonomy. The depth differences are normalized dividing them by the taxonomy depth, so that a depth difference of 1 in a detailed taxonomy has less influence than a difference of 1 in a coarser one. The depth differences for all pairs of senses of two words are weighted according to the probabilities of both senses and added together.

Word distance for two synonyms is also computed over all their senses, weighted according to their probability.The distance between two words, denoted by $dist(w_1, w_2)$, is defined as either their synonym distance, if they are synonyms, or else their hyperonym distance.

### 3.2.4. Word Alternatives

Alternatives for a word are required during query expansion. They are computed considering the union $W$ of all synsets containing the word $w$. The set of alternatives for $w$ is defined as:

$$\{ s \in W \mid dist(w, s) < th \}$$

where $th$ is a fixed ceiling, useful to avoids cases where a synonym of $w$ has meanings not typically related to $w$ (e.g. "*machine*" for "*computer*").

## 3.3. Named Entity Tagger

The NE tagger in Minipar can achieve high precision in NE recognition, since it is dictionary-based, but it has limitations (for instance it does not handle unknown names). Therefore we integrated it with an external tagger, based on a maximum entropy probabilistic approach [3] that uses both a part-of-speech tagger (TreeTagger [10]) and a gazetteer to determine word features.

The Named Entity extractor identifies person names, organizations, locations, quantities and dates, and assigns to them one of the semantic types as defined in MUC [4].

To maintain uniformity of treatment, the tags produced by the NE tagger are integrated within the same tree produced by Minipar as additional semantic features for the corresponding words.

# 4. Question analysis

Question analysis extracts or identifies the following information from the question:

- the keywords to be used in the paragraph search;
- the expected answer type;
- the location of the answering entity.

These pieces of information correspond to three successive steps in the process of question answering: keyword based retrieval, paragraph filtering based on the expected answer type, and logical relation matching between questions and answer paragraphs.

## 4.1. Keyword Extraction

The first step selects words from the question for generating a suitable paragraph query.

PiQASso considers the adjectives, adverbs, nouns and verbs in the question, excluding words from a list, determined experimentally, which includes:

- nouns such as "*type*", "*sort*", "*kind*", "*name*", frequently occurring in questions but unlikely to occur in answers;
- generic verbs like "be", rhetorical ones like "*call*", auxiliary verbs;
- adjectives that qualify "*how*" (as in "*how long*", "*how far*", etc.).

Words to which the parser does not assign the part-of-speech tag are discarded: including them did not have a clear effect on performance, according to our experiments.

## 4.2. Question Classification

The *expected answer type* is the semantic type of the entity expected as the answer to the question. The expected answer type is helpful for factual questions, but not for questions that require complex explanations.

Irrelevant sentences can be often discarded simply by checking whether they contain an entity of the expected type. The TREC 2001 QA main task requires answers shorter than 50 characters and this entails that only factual questions are asked (no long explanations can be returned as answers).

PiQASso uses a coarse-grained question taxonomy consisting of five basic types corresponding to the entity tags provided by Minipar (`person`, `organization`, `time`, `quantity` and `location`) extended with the 23 WordNet top-level noun categories. The answer type can be a combination of categories, like in the case of "*Who killed John Fitzgerald Kennedy*?", where the answer type is `person` or `organization`. Categories can often be determined directly from a wh-word: "*who*", "*when*", "*where*".

The category for "*how <adjective>*" is determined from the adjective category: "*many*", "*much*" for quantity, "*long*", "*old*" for time, etc.

The type for a "*what <noun>*" question is normally the semantic type of the noun, as determined by WNSense. For instance in "*what king signed the Magna Charta*?", the semantic type for "*king*" is person. When feasible, the WordNet category for a word is mapped to one of the basic question types. The other cases are mapped to one of the top-level WordNet categories by means of WNSense: "*what metal has the highest melting point*" has the semantic type "*substance*".

For "*what <verb>*" questions the answer type is the type of the object of the verb. "*What is*" questions, which expect a definition as an answer ("*what is narcolepsy*?", "*what is molybdenum*?") are dealt specially. The answer type is the answer itself (a disease, a metal). However, it is often not possible to just look up the semantic type of the word, because lack of context does not allow identifying the right sense. Therefore, we treat definition questions as type-less questions: entities of any type are accepted as answers (skipping the semantic type filter), provided they appear as subject in an is-a sentence.

## 4.3. Proper and Common Names

Questions whose expected answer type is `person` require special treatment. A question like "*who is Zsa Zsa Gabor*?" expects a definition, and therefore a common noun as an answer, while a question like "*who is the king who signed the Magna Charta*?" expects a proper noun. Therefore the rule for the case of a `person` answer type is: if the question contains a proper noun, a common noun is expected and vice versa.

## 4.4. Relations between Words

Relations between words in the question are determined from the dependency tree built by Minipar. In a question like "*who killed John F. Kennedy*", Minipar identifies the verb "*killed*" as having "*Kennedy*" as object, and a missing subject (represented by an empty node – a node with no corresponding word). In a possible answer sentence the verb "*kill*" may appear with exactly "*Kennedy*" as object. However the same relation could also be stated in a quite different syntactic form, where the dependencies are not so explicit, but require more complex analysis of the tree, as discussed later.

### 4.4.1. Identifying the Answer Node

In the dependency tree of the question we must identify the node that represents the object of the question. We call this the *answer node*, since it can be considered as a placeholder to be matched with the answer object in the answer paragraph. The answer node will have the answer type as determined above. Often this node exists and is empty: it corresponds to the missing subject of a verb, as in "*who killed John F. Kennedy*". In other cases the node is not empty: for instance in "*What instrument did Glenn Miller play?*", the answer node corresponds to the word "*instrument*". The answer type of the question is "*artifact*" and the semantic type of the word "*instrument*". In a direct answer like "*Glenn Miller played trombone*" the answer entity ("*trombone*") occurs in the place held by the word "*instrument*" in the question.

| Head | Relation | Modifier |
|---|---|---|
| play | obj | instrument |
| play | s | Miller |
| play | s | Glenn |
| Miller | lex-mod | Glenn |

**Table 2: relations for the sentence "*what instrument did Glenn Miller play*?"**

By experimenting with a number of questions and analyzing Minipar output, we noticed that the answer node often corresponds to the first empty subject node in the question. This is because the main verb in a question is often the first one, and because an empty subject means that the actual subject is missing.

An exception to this rule is when the required entity does not participate in the action as a subject, as in question "*in what year did the Titanic sink*?". In this case, the answer is a complement, and the answer node is still in relation with the verb, but as a complement instead of as a subject. In such cases, there is no such node in the output of the parser, and a new one must be created.

These simple heuristics are effective for simple questions: dealing with more involved expressions will require extending such heuristics, since determining the answer node is a critical issue in our approach.

# 5. Query Formulation and Expansion

## 5.1. Query formulation

The first iteration in the question answering process performs keyword extraction and query formulation.

A keyword search is performed for selecting candidate answer sentences from the whole Tipster document collection. Further iterations perform various level of query expansion, each allowing larger recall in the search.

PiQASso only addresses the problem of finding answers that are fully justified within a single paragraph. This simplifies textual analysis, as only one sentence at a time needs to be analyzed.

Although sentence boundaries information is stored in the index, search is performed document-wise, in order to achieve better recall. Consider two sentences like: "*Neil Armstrong walked on the moon. Armstrong was the first man to walk on Earth's satellite*". A question like "*Who was the first man to walk on the moon*?" would yield the keywords "*first*", "*walk*", "*moon*" and "*man*". However, while the two sentences contain all those keywords, none of them does by itself. Instead of looking for keywords within a each individual sentence, PiQASso performs a proximity search, looking for terms within a specified word position distance. For the above example, the query could be:

```
proximity 100 ((first) & (walk*) & (moon))
```

which looks for the term "*first*", for the prefix "*walk*" and for the word "*moon*" within a window of one hundred words. Such window would spans across the two sentences above, which would be both returned, individually, as candidate answers.

Keyword expansion would hardly propose "*satellite*" as an alternative to "*moon*", and therefore the second sentence would not be returned if paragraphs had been indexed separately. When evaluating the second sentence within the last filter, the match between "*moon*"

and "*satellite*" will be given a certain distance as hyperonym, allowing the paragraph to pass the filter.

We use the following criterion for choosing the size of the proximity window. In principle question and answer lengths are not strictly related: an answer to a short question may appear within a very long sentence, and vice versa, an answer to a complex question could be much shorter than the question itself. However, it seems reasonable to expect that the keywords in an answer paragraph are not too spread apart. The size of the proximity window is twice the number of nodes in the question parse tree (including irrelevant or empty nodes that may account for complicated sentences).

The heuristics proposed for keyword extraction is adequate for short questions, but returns too many terms for long questions. Thus, we split keywords into two sets: those that must appear within the proximity window and those that must occur anywhere in the document. The generated query consists of a conjunction of those terms that must be found in the document, and of a proximity query.

Terms are put in either of the two sets depending on the distance of the term from the root of the parse tree. Terms closer to the root, and therefore more central to the question, are required to be appear within the proximity window, while the others are accessory: they are only requested to occur in the document, but may be missing from the sentence.

## 5.2. Query Expansion

The first expansion step tries to cope with morphological variants of words by replacing each keyword with a prefix search, obtained from stemming the original word. Certain prefixes that appear frequently in questions are discarded: for instance "*locate*", "*find*", "*situate*" in questions expecting a location as an answer, "*day*", "*date*", "*year*" in questions expecting a date and so on. We use about a dozen of such exceptions, which correspond to cases in which the type makes these words superfluous.

Stemming is performed using Linh Huynh implementation of *Lovins's stemmer* [6].

In the second expansion cycle we broaden the search by adding (in `or`) the *synonyms* of the search terms. Synonyms are looked up in WordNet by means of WNSense. Synonyms are stemmed as well.

In the third and fourth expansion cycles, we increase recall by dropping some search terms. During the third cycle, adverbs are dropped.

During the last expansion cycle, if the query contains more than three keywords in conjunctive form, verbs are discarded, as well as person's first names when the last name is present. If after such a pruning there are still more than three keywords in `and`, then we also drop those keywords whose parent (as from the dependence tree) is already within the keywords to be searched. This has the effect, if looking for a "black cat", to perform a search for a "cat", black being a modifier of cat, and therefore depending on it.

# 6. Type Matching

The sentences returned by the query are analyzed and checked for the presence of entities of the proper answer type, as determined by question analysis.

Sentences are parsed and recognized entities are tagged. The tree is then visited, looking for a node tagged with the expected answer type.

We also check whether an entity which occurs in the sentence is already present in the question. A question like "*Who is George Bush's wife*?" expects a proper person name as an answer. The sentence "*George Bush and his wife visited Italy*" contains a proper person name, but does not answer the question. Such sentences occur frequently (the search keywords being "*George*", "*Bush*" and "*wife*"), so it is convenient to discard them as early as possible.

Sentences not verifying this condition are rejected.

# 7. Relation Matching

Sentences that pass the answer type filter are submitted to the relation matching filter, which performs a more semantic analysis, verifying that the answer sentence contains words that have the same type and relation than corresponding words in the question.

The filter analyzes Minipar output in order to:

- determine a set of relations between nodes present both in the question and in the sentence;
- look for relations in the answer corresponding to those in the question;
- compute the distance of each candidate answer and select the one with the lower distance.

In order to simplify the process, not all the nodes in the question and in the answer are considered. The same criterion used for selecting words as search keywords is applied also in this case: nouns, verbs, adjectives and adverbs are relevant (including dates and words with unknown tag).

During this analysis, the parser tree is flattened into a set of triples (*H*, *r*, *M*): head node, relation, modifier node. This representation is more general and allows us to turn the dependency tree into a graph.

In fact it is often useful to make certain relations explicit by adding links to the parser tree. For instance in the phrase "*Man first walked on the moon in 1969*", "*1969*" depends on "*in*", which in turns depends on "*moon*". According to our criterion, "*in*" is not a relevant node and so it will not be considered. We can however short circuit the node by adding a direct link between "*moon*" and "*1969*". More generally, we follow the rule

that whenever two relevant nodes are linked through an irrelevant one, a link is added between them. Similarly, since the NE tagger recognizes "*1969*" as a date, it is convenient to add a link between the main verb ("*walk*") and the date, since the date modifies the action expressed by the verb.

## 7.1. Extracting Relations

Questions and answers are analyzed in order to determine whether relations present in a question appear in a candidate answer as well.

PiQASso exploits the relations produced by Minipar and infers new relations applying the following rules:

**Direct link** if *B* is a child of *A* in Minipar output, *A* is related to *B* according to their relation in the parse tree.

**Conjunctions** Relations distribute over conjunctive links. For example, in "*Jack and John love Mary*", the relation between John and Mary is distributed over the `conj` link between John and Jack, i.e. a "*love*" relation between Jack and Mary is inferred.

**Predicates** *A* and *B* are related if they are both child of a "*to be*" verb, the first with the role of subject, the second as predicate (which is the relation between them). This is because a question like "*who is the Pope*?" is often answered by phrases such as "*The Pope, John Paul II, ...*" in which the answer does not go through a "*to be*" verb.

**Possession** *A* and *B* are related with the relation of genitive if *A* is the subject of a verb "*to have*" and *B* is the object. The rule enables matching "*John's car*" with "*John has a car*".

**Location** *A* and *B* are in inside relation if there is a `subj-in` relation between *B* and *A* (phrase of the form "*A is in B*"). This allows matching "*Paris is in France*" with "*Paris, France*".

**Invertible relations** Some relations are invertible, so that *A* and *B* are in relation if *B* and *A* are in relation as either apposition (a particular case of nominal compound) or by the `person` relation (a relation between the first and second name of a person).

**Dates** Minipar links a modifier (e.g. "*in 1986*") to the closest noun: a relation between the main verb (describing the action) and the date is inferred.

**Empty nodes** Empty nodes represent an implicit element of the sentence. Minipar sometimes can determine the word they refer to: in this case we add a relation between *A* and *B* if there is a relation between *A* and *C*, and *C* is an empty node referring to *B* (and dually for the first node in the relation).

**Negation** A relation between two nodes is discarded if both nodes depend from a node with a negative modifier. This accounts for negative phrases like "*John is not a policeman*" and avoids inferring a relation between "*John*" and "*policeman*". This rule has precedence over the others.

## 7.2. Finding a match

Suppose the following relations appear in a question: $qr_1 = (A, r_1, B)$ and $qr_2 = (A, r_3, C)$. Suppose the following relations are present in a candidate answer sentence: $ar_1 = (1, R_1, 2)$, $ar_2 = (2, R_2, 3)$ and $ar_3 = (1, R_3, 3)$. All matches between triples in the question and in the answer are considered, provided that no node is put in correspondence with two different nodes: if we match $qr_1$ with $ar_1$, we cannot match $qr_2$ with $ar_2$, or node *A* in the question would have to match both nodes 1 and 2 in the answer.

The match with the smallest distance is selected.

## 7.3. Matching Distance

An answer paragraph can be considered as a close answer to a question if it contains nodes and relations corresponding to all nodes and relations in the question. For each missing node the distance is increased by an amount that depends on the relevance of the node. To represent this relevance we associate a *mismatch distance*, $mmd(n)$, to each node $n$ in the question. For instance the mismatch distance is small for the node corresponding to the question type (e.g. the node "*instrument*" in a previous example), since it may be missing in the answer. Nodes depending on other relevant nodes have half the mismatch distance of their parents: they may express a specification that a correct answer need not contain.

The overall *matching distance* between a question and a candidate answer is computed by summing, for each node $n$ in the question:

$$\begin{cases} mmd(n) \cdot dist(n, m) & \text{if } n \text{ matches node } m \text{ in the answer} \\ mmd(n) & \text{otherwise} \end{cases}$$

and similarly for each relation in the question.

The distance is incremented to account for special situations, e.g. when the answer is too specific: for the question "*Who was the first man in Space*?", "*The first American in space was …*" is not a proper answer, contrary to a naïve rule that wants a specific answer correct for a general question.

If the answer sentence does not contain an entity of the expected answer type, the distance is set to infinity, to ensure that the paragraph is rejected.

The maximum distance from the question allowed by the filter (distance ceiling) may be set to either: a) very high, so that any sentence matching the answer node will

pass the filter (recall-oriented); or b) proportional to the number of nodes in the question. PiQASso implements a simple tightening strategy whereby the ceiling is decreased at each expansion iteration, avoiding too much garbage in the early phases.

## 8. Answer Popularity

After the TREC 2001 submission we introduced a criterion for selecting answers based on a measure of answer popularity, which proved quite effective.

Answers are grouped according to the value contained in their answer node. A score is assigned to each group proportional to the average of the matching distances in the group and inversely proportional to the cardinality of the group. Groups are sorted by increasing score value and only the answer with the smallest matching distance in each group is returned.

The criterion combines a measure of difference to the question and a measure of likelihood based on how often the same answer was offered. Selecting only one answer per group ensures more variety in the answers.

## 9. Results

PiQASso achieved the scores summarized in Table 3, expressed as MRR (Mean Reciprocal Rank) of up to five answers per question. The official score, computed from the judgments of NIST assessors at TREC 2001, ranks PiQASSo in 15th overall position. PiQASso achieved the same score for both strict evaluation (answers supported by a document in the collection) and lenient evaluation (answer not supported), since it does not use any external source of information. The unofficial score was computed by our own evaluation of the results on a new run of the system after the addition of the popularity ranking.

| Run | MRR Strict | MRR Lenient |
|---|---|---|
| TREC 2001, official | 0.271 | 0.271 |
| TREC 2001, unofficial | 0.32 | |

**Table 3: Scores in the TREC 2001 QA main task.**

A peculiarity of the TREC 2001 questions was the presence of a higher than usual percentage (almost 25%) of definition questions that could have been answered by simple lookup in a dictionary or from other sources (e.g. the Web), as some other systems did. For PiQASso we concentrated in improving the system ability to analyze and extract knowledge from the given document collection.

## 10. Assessment

In order to assess the effectiveness of the various filters, and how they affect the overall performance we performed some measurements using a subset of 50 questions of the TREC 2001 set. Results are summarized in Table 4. For half of the questions (49%), no paragraph passed all filters. The great majority of answers are obtained from the results of the first IR query.

| | % |
|---|---|
| Questions for which no answer was found | 49 |
| Questions answered by first query (over all answers) | 92 |
| Questions for which no paragraph was retrieved | 2 |

**Table 4: Filter effectiveness.**

The benefits of iterating the process after performing query expansion are less than expected.

Overcoming this limit requires improving query expansion to produce more word alternatives or morphological variations. This may however complicate the task of the relation matching filter, which also needs to be refined: the paragraphs retrieved by the initial query (without stemming or synonym expansion) are simpler to match with the question and produce most of the answers. When more complex paragraphs are retrieved by the more complex queries, matching is more difficult and rarely an answer is found.

The current system is not capable, for example, of matching the sentences "*John loves Mary*" and "*John is in love with Mary*", since their main verbs "*love*" and "*to be*" are different. Either deeper semantic knowledge would be required or a collection of phrase variants, that might be built automatically with the method suggested by Lin [10], discovering similarities in paths within the dependency graph of the parser.

## 11. Conclusions and Future Work

PiQASso is engineered as a vertical application, which combines several libraries into a single application. With the exception of Minipar and WordNet, all the components in the architecture were built by our team, including the special purpose paragraph indexing and search engine, up to the tools for lexical analysis, question analysis and semantic filtering.

PiQASso is based on an approach that relies on linguistic analysis and linguistic tools, except for passage retrieval, where it exploits modern and efficient information retrieval techniques. Linguistic tools provide in principle higher flexibility, but often appear brittle, since implementations must restrict choices to reduce the efffects of combinatorial explosions. One way to improve their performance would be by providing them with large amounts of semantic data in a preprocessed form: for instance generating a large number of variants from the phrases in the document collection and matching them with effective indexing techniques and

statistical estimates, rather than performing sophisticated matching algorithms.

PiQASso is heavily dependent on Minipar since it relies on the dependency relations it creates. Such relations are often too tied to the syntactic form of the sentence for our purposes, so we had to add specific processing rules to abstract from such representation and to work around certain of its idiosyncrasies.

Question analysis could be improved by adopting a finer-grained taxonomy for the expected answer type. Such granularity requires support by the named entity tagger.

Keyword extraction/expansion would benefit from a better identification of the sense of a word, so that fewer and more accurate alternatives can be used in the query formulation. Current figures show that present keyword expansion is not effective, for it either does not add results, or it adds too many, returning way too many hits for the system to analyze them all. As for about half of the questions our system did not find any answer at all (which gives us outstanding improvement margins), this seems a necessary step.

## Acknowledgements

Alessandro Tommasi is supported by a PhD fellowship from Microsoft Research Cambridge.

Cesare Zavattari contributed to various aspects of the implementation, in particular of the sentence splitter.

## 12. References

[1] G. Attardi, A. Cisternino, Reflection support by means of template metaprogramming, *Proceedings of Third International Conference on Generative and Component-Based Software Engineering*, *LNCS*, Springer-Verlag, Berlin, 2001.

[2] D. Lin, LaTaT: Language and Text Analysis Tools, *Proc. Human Language Technology Conference*, San Diego, California, March 2001. http://hlt2001.org.

[3] A. Berger, S. Della Pietra, and M. Della Pietra, A Maximum Entropy Approach to Natural Language Processing. *Computational Linguistics*, 22(1), 1996.

[4] R. Grishman and B. Sundheim, Design of the MUC-6 evaluation. In NIST, editor, *The Sixth Message Understanding Conference (MUC-6)*, Columbia, MD. NIST, Morgan-Kauffmann Publisher, 1995.

[5] S. Harabagiu, Moldovan et al., FALCON: Boosting Knowledge for Answering Engines. *TREC 2000 Proceedings*, 2000.

[6] J. B. Lovins, Development of a Stemming Algorithm. *Mechanical Translations and Computational Linguistics*, 11: 22-31, 1968.

[7] G. Miller, Five papers on WordNet. *Special issue of International Lexicography* 3(4), 1990.

[8] J. C. Reyner and A. Ratnaparkhi, A Maximum Entropy Approachg to Identify Sentence Boundaries. *Computational Language*, 1997.

[9] G. Schmid, TreeTagger – a language independent part-of-speech tagger, 1994. Available: http://www.ims.uni-stuttgart.de/Tools/DecisionTreeTagger.html.

[10] D. Lin and P. Pantel, Discovery of Inference Rules for Question Answering. To appear in the *Journal of Natural Language Engineering*, 2001.