TRESONTANI Quentin
Student ID: 20405036
Exchange Student

COMP 5211 - Advanced Artificial Intelligence

2016 Fall – Project report

November, 21st 2016.

Analysing, Classifying and Solving Winograd Schemas based on Inference Classification, Natural Logic, Natural Language Processing and Common Knowledge.

Project Repository: *https://github.com/Qww57/winosolver*

Introduction

Artificial Intelligence could be defined as the development of algorithms copying human design thinking patterns in order to obtain automatic complex reasoning. However, quantifying and testing the performance of AI systems isn't always obvious. Since 1950, the reference was the Turing test [1]. The Winograd Challenge [2] has been created as an alternative to the Turing test which has been criticized for its subjectivity and for its relation to the creation of a personality for the program being tested. The Winograd Challenge is based on a set of schema that should be solved by the candidate software and whose success rate is then objective and computationally easy to obtain.

This report focuses on analysing, recognizing and proposing a solution for one category of schemas named Direct Clausal Events (shortened as DCE). The first part analyzes the structure of Winograd Schemas. It discusses their similarities with textual entailment problems, and how to classify schemas depending on the way of solving them. The second part of the project focuses on feature definition and feature extraction for schema classification based on the example of DCE schema. Initial objective definition and classification result are discussed in detail. Third part of the report focuses on proposing a simple solution to solve DCE schemas based on Natural Logic. Strength and weakness of this solution are discussed based on resolution examples.

I. Analysis of the different Winograd schema

Most of the examples discussed in this report are taken from the XML file created by the New York University [3].

A. Structure of a Winograd Schema

Structure - A schema is composed of a pair of sentences which only differ by one specific element. Both of these sentences contain a pronoun whose antecedent reference in the sentence is ambiguous and dependent of the differing element. In order to solve the Winograd schema, one has to choose the right answer between the two possibles references proposed as answers. The pronoun in each of the sentence has a different referent that can be uniquely determined.

Winograd Schema n°1-2:
The city councilmen refused the demonstrators a permit,
because **they (advocated / fear)** violence.

In this example, it is easy for human to determine that "they" refers to the "city councilmen" in the first sentence whereas it refers to "the demonstrators" in the second one.

In Winograd schemas, both potential referents have the same gender and number than the pronoun. The pronoun referent cannot be guessed on a purely grammatical analysis. Both answer possibilities should also be realistic in order to avoid too simple problems.

The women stopped taking pills because they were **(carcinogenic / pregnant)**.
Who was **(carcinogenic / pregnant)**?

This example is not considered as a valid Winograd schema since pills cannot get pregnant and women cannot be carcinogenic.

Terminology - A schema is considered to be composed of the following elements.

- Sentence: one of the two sentences used in the schema.
- Snippet: part of the sentence containing the differing element which is the key to solve the schema.
- Basis: common part of the two sentences, it usually does not containing the snippet.
- Ambiguous pronoun: pronoun whose reference has to be disambiguated by the program.

These terms will be used in the next parts of the report.

B. Winograd Schema as Textual Entailment Recognition problems

This section discusses the similarity of Winograd schemas with entailment and inference problems. Based on this similarity, a classification of Winograd schemas is discussed.

1. Inference and entailment

Inference - Inference is the reasoning process of drawing a conclusion C based on some premises or knowledge KB. It can then be seen as an educated guess.

Entailment - A sentence C is entailed by a knowledge base KB if the sentence S is true in all interpretations in which the knowledge base KB is true.

KB	***C***	***C inferred from KB***	***KB entails C***
True	True	True	True
True	False	False	False
False	True	True	*Not applicable*
False	False	True	*Not applicable*

The concept of inference is logical stronger than the one of entailment. Having an inference implies then having entailment. In the following parts will try to match Winograd Schema solving to textual entailment.

2. Recognizing Textual Entailment (RTE)

According to the book *Natural Processing with Python* [4], “Recognizing Textual Entailment (RTE) is the task of determining whether a given piece of text entails another text called hypothesis”. RTE is a classic problem of natural language processing that slightly differs from pure logical entailment. “RTE is not intended to be purely logical entailment, but rather whether a human would conclude that the text provides reasonable evidence for taking the hypothesis to be true”.

The following example shows a textual entailment recognition problem:

> ***T****: Parviz Davudi was representing Iran at a meeting of the Shanghai Co-operation Organisation (SCO), the fledgling association that binds Russia, China and four former Soviet republics of central Asia together to fight terrorism.*
>
> ***H****: China is a member of SCO.*

In this example, based on the text T, the hypothesis H is likely to hold.

3. From Winograd Schema to RTE

Winograd schema can be transformed into problems similar to textual entailment ones.

Winograd Schema n°1-2:
The city councilmen refused the demonstrators a permit,
because they **(fear/ advocate)** violence*.*
Who **(fear/ advocate)** violence?

From the first case (with the verb *fear*), a piece of text T and two hypotheses H1 and H2 can be defined. A similar set T, H1 and H2 can also be obtained with the verb *advocate*.

> ***T:*** *The city councilmen councilmen refused the demonstrators a permit because they fear violence.*
>
> ***H1:*** *The city councilmen fear violence.*
> ***H2:*** *The demonstrator fear violence.*

Whereas RTE problem would only focus on verifying if T entails H, solving a Winograd schema, would be to verify which one of the two hypotheses is the most likely to hold. From this similarity, it can be guessed that techniques used to solve textual entailment problems can also be considered in order to solve Winograd Schemas. These techniques usually involve semantic analysis, grammatic analysis, extraction of knowledge and use of common knowledge. The third part of this report will especially discuss how to apply one solution for textual entailment to Winograd Schemas to solve them.

In the context of the Winograd Challenge, the truth value of the sentences are not discussed. It is then assumed that the sentence T is always true. In this context, it is possible to refer to Winograd Schemas as textual entailment or as textual inference without distinction.

C. Classification of Winograd schema by inference type

1. Inferences and ways of reasoning

Four main ways of reasoning can be defined to conduct inferences. Deduction is the process of reasoning on facts and rules in order to get conclusions. Induction is the process of reasoning on facts and conclusion in order to get rules. Abduction is the process of reasoning on rules and conclusion in order to get facts. Probabilistic reasoning is the process of reasoning with uncertainty based on probabilities related to facts.

Resolving Winograd schemas uses all these kinds of reasoning depending on the schema and on the way chosen to solve it.

Metz Football Club has defeated Paris because it was **(better / weaker)**.
Who was **(better / weaker)?**

Only the first sentence will be discussed since the same conclusions applies to the other one.

In order to solve this schema the additional knowledge that "better teams are more likely to win" is needed. This sentence looks like a general rule. For simplification issue, the uncertainty issue inherent to sports is neglected here. A fact, a rule and a conclusion can be then distinguished:

- *Fact*: It is better.
- *Rule*: The better team wins.
- *Conclusion*: Metz Football Club has defeated Paris.

From the conclusion and the rule, it can be inferred that Metz was better and then *it* refers to Metz Football Club. As a consequence, solving this schema is abductive reasoning.

However, the missing knowledge could have been expressed as, "teams winning are usually better". *Metz has defeated Paris* is then considered as a fact and the schemas can be split in:

- *Fact*: Metz Football Club has defeated Paris.
- *Rule*: Teams winning are usually better.
- *Conclusion*: It is better.

From the fact and the rule, it can be inferred again that Metz was better. Thus, deductive reasoning has been used to solve the schema.

2. Different kind of inferences

Different types of inferences can be distinguished. This distinction is based on the type of problem to solve and the required reasoning capabilities. Among others, these types of reasoning are frequent in natural language:

- Spatial reasoning: *The trophy is on the table* *The table is under the trophy*
- Causal reasoning: *Mary forgot the time the library closes* *Mary knew it*
- Paraphrase reasoning: *Mary took a flight to New York* *Mary flew to New York*
- Relation extraction: *Mary's dog is named Mickey* *Mickey's master is Mary*

As a consequence, these types of reasoning are also present inside Winograd Schemas. Two common ones are:

- Spatial reasoning:

Winograd Schema n°21-22:
Tom threw his schoolbag down to Ray after he reached the **(top / bottom)** of the stairs.
Who reached the **(top / bottom)** of the stairs?

Winograd Schema n°3-4:
The trophy doesn't fit into the brown suitcase because it is too **(large / small)**.
Who is too **(large / small)**?

Solving these schemas implies the model to have space representation abilities as in the schemas 21-22 or some sizing capacities as in the schemas 2-3.

- Causal reasoning:

Winograd Schema n°1-2:
The city councilmen refused the demonstrators a permit,
because **they (advocated / fear)** violence.

Solving these schemas implies the model to draw conclusions based on elements of the sentence and elements of the knowledge database.

We can note that paraphrase reasoning will implicitly be used in most of the resolution processes in order to match knowledge extracted from the sentence with the information available in the database. Thus, each schema can involve various types of reasoning and can belong to many of these mentioned inference classes.

3. Direct Clausal Event

Among all these types of schemas, this part focuses on schema related to causal reasoning and especially, on some schema that will be described as Direct Clausal Event. In *"An Approach to Solve Winograd Schema Challenge Using Automatically Extracted Commonsense Knowledge*" [5], Direct Clausal Event (also named Event-event-causality) is described as follows:

> *"In this category, the commonsense knowledge required for resolution has two mutually causal events (explained and convince in the example below) such that a pronoun participates in one of the event and its candidate co-referent participates in another."*

In order to be even more restrictive, DCE schemas in this report are redefined as possessing a structure (mostly containing only two main proposition). Among the 282 Winograd Schemas available, 54 of them have been recognized as matching this description.

The schemas are those numbered: *1, 2, 7, 8, 9, 10, 11, 12, 15, 16, 23, 24, 29, 30, 31, 32, 35, 36, 39, 40, 45, 46, 51, 52, 69, 70, 89, 90, 135, 136, 147, 148, 151, 152, 211, 212, 215, 216, 227, 228, 253, 254, 255, 260, 261, 262, 263, 264, 265, 268* and *269*.

Conclusion of the analysis:

Winograd schemas are based on two sentences containing a pronoun whose reference is ambiguous. These sentences are only differing by one word that is essential to determine the pronoun reference. Grammatical analysis is not sufficient to find the pronoun reference and common knowledge has to be used. We have shown that it is possible to transform Winograd Schemas into textual entailment recognition problems which means that RTE's solution proposals might then be used for Winograd Schema as well. This similarity with entailment problems has also helped us to discuss various types Winograd Schemas based on the underlying ways of reasoning (abductive, deductive…) and on inference types (spatial, causal…). Aiming to solve a broad set of Winograd Schemas, implies then to construct a model able to deal with all these cases. In order to simplify the problem, one specific type of schema named DCE has been chosen. The objective is now to be able to detect them and to solve them using textual inference techniques.

II. Classification of Winograd Schema

A. Problem description and initial objectives

Description of the problem - From an handmade classification of schemas, a set of labels can be created. The objective of the classifier is then guessing labels representing the class belonging representing the underlying inference type of the schema. Since Winograd Schemas can involve many types of inferences, they should be able to have many class-labels. This problem is then a multilabel classification problem.

As a first approach to Winograd Schemas classification, this report focuses only on designing a binary classifier for DCE schemas. The classifier is based on the list of DCE schemas mentioned previously. In order to integrate it in the multilabel classification problem, this classifier will be cooperating with other class dedicated classifiers. "All-vs-All" or "One-vs-One" predicting models can be then used.

This section discusses feature extraction based on the Winograd sentence number 1:

The city councilmen refused the demonstrators a permit because they feared violence.

Initial objective metrics - The initial objective of this project is to achieve specific accuracy, specificity and sensitivity rates. As a reminder, these metrics are computed based, in the DCE classification case, on the numbers of:

- *True Positives (TP) :* DCE classified as DCE
- *True Negatives (TN):* Non-DCE classified cted as non-DCE.
- *False Positives (FP)*: Non-DCE classified as DCE.
- *False Negatives (FN):* DCE classified as non-DCE

The three metrics are then defined as follows:

- Accuracy = (TP + TN) / (TP + TN + FP + FN)
- Sensitivity = TP / (TP + FN)
- Specificity = TN / (TN + FP)

Accuracy **-** Accuracy is usually one of the most representative metrics for classification since it shows the frequency of correct classification guesses. However, it can be biased by an unequal distribution of the labels. This is known as the class imbalance problem. In the corpus of schemas, 54 of them among 273 are classified as DCE, corresponding to a frequency of 0.197. A classifier with a tendency for non-DCE classification would then be able to achieve a high accuracy that is only reflecting the class distribution. For instance, a classifier returning only non-DCE prediction would achieve an accuracy around 0.8 without helping to detect DCE schemas. As a consequence, the accuracy objective should be higher than the non-DCE distribution in order to discriminate bad classifiers. The minimum accuracy objective has then been defined as 0.9.

***Sensitivity* -** Sensitivity focuses on the ability of the classifiers to distinguish DCE as DCE. High sensitivity implies that the classifier is not missing any DCE. This is really important in this project. The initial objective is then of 0.9. Getting to such a result will be challenging.

***Specificity* -** The specificity indicator describes the ability of the system to classify non-DCE as non-DCE. However, in the context of a future cooperation with other class-dedicated classifiers in "All-vs-All" or "One-vs-One", a lower sensitivity would be balanced by the other classifiers votes. The objective is then of 0.8.

Indicator	***Minimum Value***
Accuracy	0.9
Sensitivity	0.9
Specificity	0.8

Once these objectives are achieved, other performance metrics such as Precision, Recall and ROC Curves can be used for further details.

B. How to choose a feature set for schema classification?

This section formalizes the properties of significant features for Schema classification.

Note: *This part discussed properties of features helping to get a better accuracy. The benefits of matching these properties will be discussed informally. However, in the context of a bigger study, a protocol should be established to perform test and validate the benefits of these properties on classification results.*

***Definition of significant features* -** A significant feature for belonging to the class C is defined as a feature describing C (a) while discriminating non-C (b). Having features for class C which are two-by-two and, eventually, mutually independent could also be interesting in some cases, but does not seems to be mandatory. Independency is discussed in the appendix.

1. Class characterization

The following three properties can be used to verify class characterization:

$P(C) < P(C \mid Ri)$	The probability of belonging to class C should be positively correlated to the respect of the rules. We should then verify for each rule Ri of the feature set that $P(C) < P(C \mid Ri)$ holds.
$P(Ri \mid C) > \alpha$	The probability of having the rule being respected in the class should be especially high. We would then need to have $P(Ri \mid C) > \alpha$. If the probability of having C respected is low, then this class might not help us.

P(DCE | non-R1) < β	The probability of having belonging to the class should be low if the rule is not respected.

2. Discrimination of other classes

P(others) > P(others | Ri)	The probability of having an other class than C should be negatively correlated to the respect of the rules. We should then verify for each rule Ri of the feature set that P(others) > P(others | Ri) holds.

As a consequence, all features will be tested using the following inequalities:

- P(Ri | C) > α
- P(C) < P(C | Ri)
- P(DCE | non-R1) < β
- P(others) > P(others | Ri)

C. Selection of a feature set for DCE classification.

This section formalizes the features of Direct Causal Event discussed earlier in the report. Major steps of the implementation of each feature are discussed. For further detail, all implementations are available in *features_tools.py* and all tests on rule probabilities are available in *rule_testing.py.*

1. Structure of a DCE schema

R1 - *Schema structure:* A DCE schema should be based on this pattern:

X action Y LL Z action.

The common structure of the schema sentence should match the following pattern:

Complement(s) X action Y complement(s) LL Z action complement(s).

Where:

- X represents the first reference possibility for the pronoun.
- Y represents the second reference possibility for the pronoun.
- Z is the nominal group containing the pronoun.
- Action represents a verb that can be stative or active. *(Action is the snippet might also be called trait)*
- Complements does not have any constraints.
- LL represents the logical link relating the snippet to basis of the schema.

Implementation:
The implementation of this feature has been done as follows:

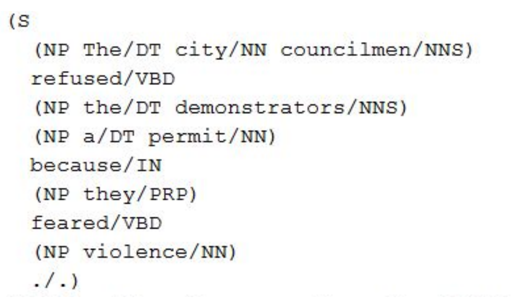
```python
def is_dce_structure(schema):
    # Basic structure of DCE schemas
    structure = ["N", "V", "N", "IN", "N", "V"]

    # Pre-processing
    full_structure = Chunker().parse(schema.sentence)
    main_structure = get_main_pos(full_structure)
    tags = [tag for (tag, words) in main_structure]
    tags = ["V" if "V" in tag else tag for tag in tags]
    tags = ["N" if tag in ["NN","NNS","NP","NPS"] else tag for tag in tags]

    # Checking if the basic structure is contained in the schema
    return is_sub_sequence(structure, tags)
```

Example : Following printings are showing the different steps of the process on the example:

Winograd Schema n°1:
The city councilmen refused the demonstrators a permit because they feared violence.

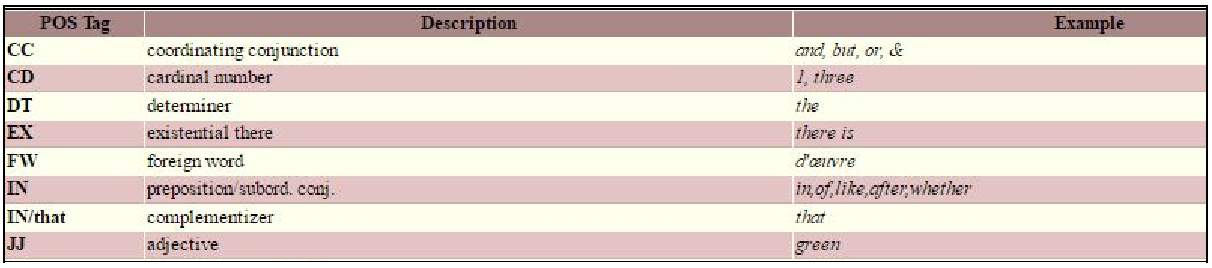
The second line is used in order to parse the sentence using a ***chunker*** and a ***parser***.

The ***parser*** is used to tokenize the sentence and recognize the nature of each token. The parser that has been used in name *TreeTagger* [6]. The structure of the sentence is represented below.



```
(S
  (NP The/DT city/NN councilmen/NNS)
  refused/VBD
  (NP the/DT demonstrators/NNS)
  (NP a/DT permit/NN)
  because/IN
  (NP they/PRP)
  feared/VBD
  (NP violence/NN)
  ./.)
```

All elements in the tree structure are now related to some tags. Below is an example of some of of tags used by TreeTagger [7].



POS Tag	Description	Example
CC	coordinating conjunction	*and, but, or, &*
CD	cardinal number	*1, three*
DT	determiner	*the*
EX	existential there	*there is*
FW	foreign word	*d'œuvre*
IN	preposition/subord. conj.	*in,of,like,after,whether*
IN/that	complementizer	*that*
JJ	adjective	*green*

The ***chunker*** is used in order to simplify the sentence structure and recognize elements as a group. For instance *"The city councilmen"* is considered as a single token inside the tree structure and not as a determiner and two nouns. The third line is used to convert the tree as a readable list of tokens and tags.

```
[('NP', 'The city councilmen '), ('VBD', 'refused'), ('NP', 'the demonstrators '),
```

Two filters are then applied in order to group the different types of tags for verbs together under the letter V and to put all the different tags of nouns together under the letter N.

```
['NP', 'VBD', 'NP', 'NP', 'IN', 'NP', 'VBD', 'NP', '.']
['N', 'V', 'N', 'N', 'IN', 'N', 'V', 'N', '.']
```

Finally, a recursive function is used in order to verify if the minimum structure of tags is included inside the sentence with respect to its ordering. The function is returning a boolean and defined as follows:

```
def is_sub_sequence(sub_seq, seq):
   if len(sub_seq) == 0:
       return True
   if sub_seq[0] in seq:
       index = seq.index(sub_seq[0])
       if index + 1 < len(seq) or len(sub_seq) > 1:
           return is_sub_sequence(sub_seq[1:], seq[index + 1:])
       else:
           return True
   else:
       return False
```

Results:

When applying this feature on the full schema set, results are:

```
P(DCE) = 0.18681318681318682
P(R1) = 0.7289377289377289

P(DCE | R1) = 0.21608040201005024
P(non-DCE | R1) = 0.7839195979899497
P(R1 | DCE) = 0.8431372549019608
P(DCE | non-R1) = 0.10810810810810811
```

Despite being a common rule *(P(R1) = 0.7)*, this rule is still satisfying the four conditions mentioned previously and should then help us classifying DCE schemas.

Implementation alternative:

This hasn't been the only way to implement this rule for classification. In some classifiers, this rule has been considered by directly returning the list of tags of the basis of the sentence in one feature and the list of tags of the snippet in another feature. This approach has shown interesting results too.

2. Causality or opposition link

R2 - *Causality or opposition:* The logical link in Direct Causal Events is representing either causality or opposition.

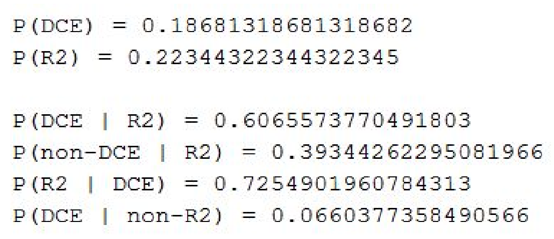
Implementation:
This is achieved by getting the link inside between the basis of the sentence and the snippet. The first line is used to delete the snippet from the sentence. The remaining part is then tokenized. Based on the tag, the logical link is returned as follows:

```
def get_link(schema):
   main_prop = get_main_prop(schema)
   str_main_prop = nltk.word_tokenize(main_prop)
   sentence = analyze(schema.sentence)
   link_set = [w.lemma for w in sentence if w.postag in ["IN", "RB", "CC",
"RBS", "RBR"] and w.word == str_main_prop[-1]]
   if link_set:
       return link_set[0]
   else:
       return ""
```

Even if this implementation might not be sufficient in some tricky cases, it has shown pretty accurate results. The following functions are then used to return booleans. *causal_set* and *oppoisition_set* are handmade sets of logical links.

```
def is_causal_relation(schema):
   return True if get_link(schema) in causal_set else False

def is_opposition_relation(schema):
   return True if get_link(schema) in opposition_set else False
```

Results:
When applying this features on the full schema set, results are:



```
P(DCE) = 0.18681318681318682
P(R2) = 0.22344322344322345

P(DCE | R2) = 0.6065573770491803
P(non-DCE | R2) = 0.39344262295081966
P(R2 | DCE) = 0.7254901960784313
P(DCE | non-R2) = 0.0660377358490566
```

This rule is then a relatively infrequent rule *(P(R2) = 0.22)* that is frequent for DCE *(P(R2 | DCE) = 0.72)*. Results seems then to be particularly interesting for DCE classification. The other properties mentioned in the previous part are also respected.

Implementation alternative:
In most of our classifiers, the analysis has also been done by directly returning the link from the *get_link* function.

3. Stative verbs in the snippet

R3 - *Causal attributive:* The verb related to the pronoun in the snippet should be a stative verb.

Some DCE schema that can be named Causal attributive schemas [5] are defined as schemas where the snipped states that the pronoun possesses a specific trait. These schemas have then a state verb in their snippet.

State verbs in English are verbs defined in opposition to dynamic verb which describes an action. Stative verbs are representing a physical state as possessing, a feeling from your senses, a mental state as emotions or preferences (liking or disliking something). Some verbs can be at the same time stative or dynamic depending on the context (to be, to have, to see…) [8].

Implementation:
The implementation is very similar to the one describe previously. The snippet is tokenized, the tokens are then tagged and lemmatized. Lemmatization is the reduction of tokens to a related common base form. For verbs it is equivalent to return their infinitive form. Based on the tags in the snippet, the lemma of the verb is returned.

```
def get_snippet_verb(schema):
   snippet = analyze(schema.snippet)
   verb_set = [word.lemma for word in snippet if "V" in word.postag]
   if verb_set:
       return verb_set[0]
   else:
       return None
```

Similarly as with logical links, two function have been created to return a boolean based on two handmade sets of stative verbs. Since some verbs can be stative or dynamic, the following function has been used to distinguish all cases:

```
def snippet_verb(schema):
   verb = get_snippet_verb(schema)
   if is_action_verb(verb) and is_state_verb(verb):
       return "A-S"
   elif is_action_verb(verb):
       return "A"
   elif is_state_verb(verb):
       return "S"
   return ""
```

Results:
These results are considering that we have R3, when the *snippet_verb* function returns *S* or *A-S*:

```
P(DCE) = 0.18681318681318682
P(R3) = 0.6043956043956044

P(DCE | R3) = 0.24848484848484848
P(non-DCE | R3) = 0.7515151515151515
P(R3 | DCE) = 0.803921568627451
P(DCE | non-R3) = 0.09259259259259259
```

The rule R3 is then pretty commons especially because of verbs that can be both static and dynamic as “to be” or “to have”. However, this rule is very characteristic to DCE. (Probability of 0.8 within DCE while only of 0.6 within all schemas).

Implementation alternatives:
This rule can the be implement as a feature in other ways. *A, A-S* or *S* can be returned as a feature. It is also possible to return directly the snippet verb.

4. Other rules that have not been considered yet

Two other rules have also been considered, but have not been implemented and tested due to a lack of time. These two rules are grammatically extending the rule R1.

R4 - *Pronoun role:* The unknown pronoun is subject of the snippet.

R5 - *Referents functions:* Potential references of the pronouns should be subject of the main part (X) and a direct object (Y).

5. Selection of features

Many combinations of the feature implementation alternatives have been used for the classifier feature set. Due to a lack of time, a protocol has not been discussed and used in order to optimize the results. The selection has been done by a process of trial and error.

Among the feature sets used, the implementation that has been showing the most interesting results is composed of these four features:

```
# Main structure of the sentence after chucking - list of tags
feature_set['sentence'] = str([tag for (tag, words) in main_structure])

# Full structure of the snippet - list of tags
feature_set['snippet'] = str(snippet.get_tag_sequence())

# Logical link used in the schema
feature_set['logical_link'] = get_link(schema)

# Lemma of the verb of the snippet
feature_set['snippet_verb'] = snippet_verb(schema)
```

The two first features cover R1, the third one covers R2 and the fourth one covers R3.

D. Implementation of classifiers and result discussion

1. Methodology for implementation and discussion of classifiers

***Class imbalance problem* -** As explained previously, the data set is composed of a significant majority of non-DCE schemas: 222 versus 54 schemas. This situation is favorable to obtain a classifier that is only reflecting the data distribution, which is something we don't want [9].

***Train and test sets class balance* -** In order to minimize this effect, two measures have been applied. First, classifiers have all been trained and tested on data containing a frequency of DCE schema (0.156 to 0.216) similar to its frequency in the full corpus (0.197). This way, comparing classifiers was more likely to describe their performances and not only the random distribution of DCE and non-DCE schema in each sets. For instance, let's consider two similar classifiers C1 and C2 returning always the non-DCE answer and tested on different sets S1 and S2, with S1 containing more DCE schemas than S2. We would then have a better prediction rate for C2 than C1, meaning a better accuracy, without having a better classifier. Dealing this way also discriminates classifiers only reflecting the data distribution since our accuracy objective has been placed significantly higher (0.9 for accuracy, 0.803 for class distribution). On the other hand, oversampling has been tried for some classifiers in order to increase the number of DCE schemas in the train set and then reduce the class imbalance.

***Cost Sensitive Learning* -** If the class imbalance problem is still persisting, another method would be to apply cost sensitive learning algorithms. The importance of misclassifying a DCE schema in comparison to misclassifying a non-DCE schema would then be increased.

***Problem requalification* -** In case of unsatisfying results, the problem should then probably be reconsidered as anomaly detection problem [10] or outlier analysis problem. In this case, different methods would be used.

2. Implementation of classifiers

Naive bayesian classifiers have been chosen for the implementation. Naive Bayes classifiers are probabilistic classifiers based on the Bayes Theorem. They are based on the assumptions that all probabilities are conditionally independent. Even if this assumption might not be true, naive Bayes usually still show interesting results with sentence classification. Later on, changing the classifier the naive Bayes classifier to a Bayesian Belief network to get rid of the independency assumption can be an improvement possibility.

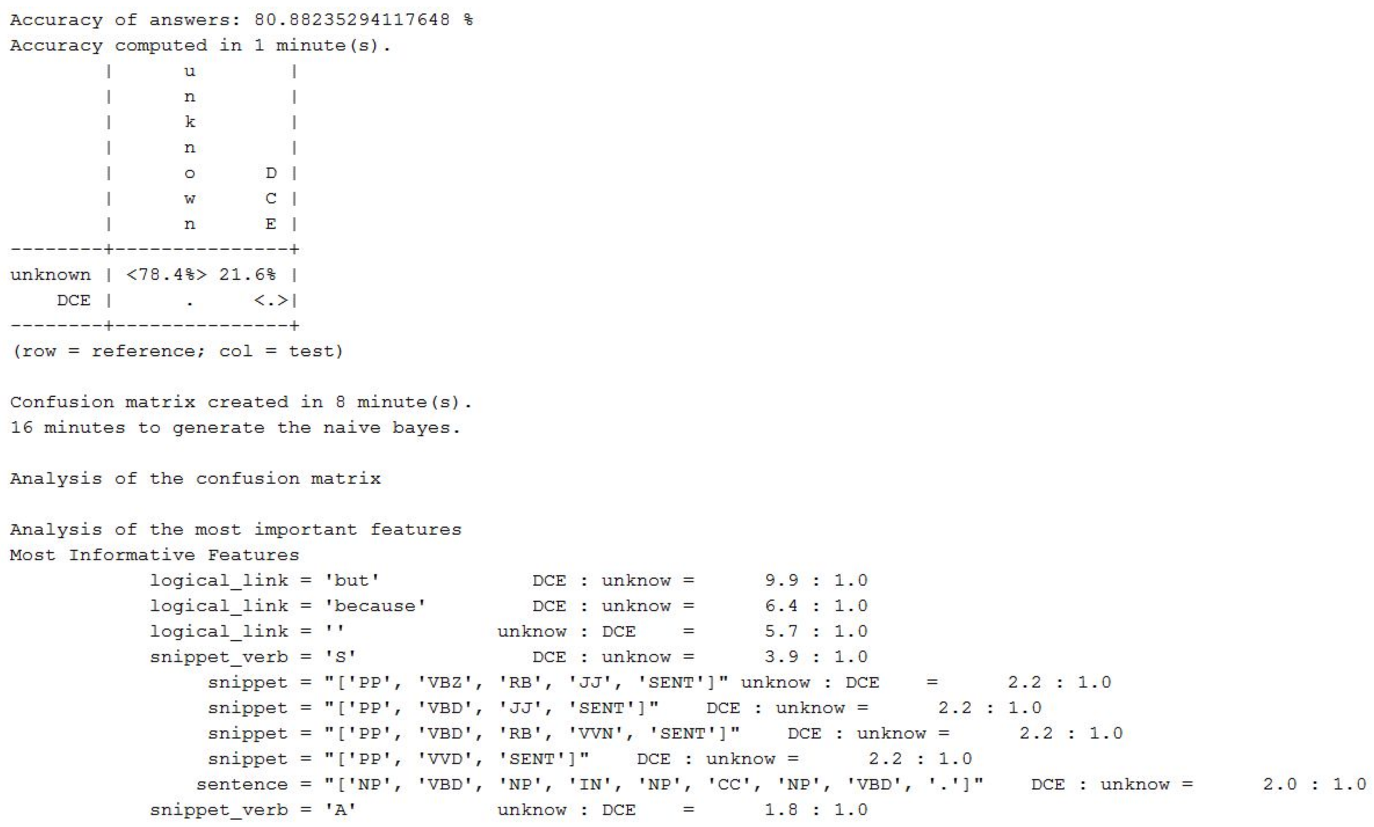
In the file named *dce_classifier.py,* a class object for classifiers has been created. The creation and training of the classifier is made when calling the class constructor. The code is pretty straightforward to read with the comments which highlight the previous discussed topics.

```
def __init__(self, classifier_type):
   self.accuracy = 0
   self.cm = "not defined"
   self.classifier_type = classifier_type
   debut = time.time()

   # Creation of the feature set
   schemes = parse_xml()
   add_labels(schemes)

   # Creating the train and test sets
   length = len(schemes) if set_length is None else set_length
   train_length = int(length * 0.5)

   # DCE frequency in corpus is around 0.197
   dce_percentage = 0
   while dce_percentage < 0.167 or dce_percentage > 0.227:
       random.shuffle(schemes)
       self.train_schemes = schemes[0:train_length]
       self.test_schemes = schemes[(train_length + 1):length]
       train_set_dce = [schema for schema in self.train_schemes
             if schema.get_type() is "DCE"]
       print(len(train_set_dce))
       dce_percentage = len(train_set_dce) / train_length

   print("Train set with " + str(dce_percentage * 100) + " of DCE schemas.")

   # Oversampling the DCE schema because of class imbalance problem
   self.train_schemes.extend(train_set_dce)
   self.train_schemes.extend(train_set_dce)
   train_set_dce = [schema for schema in self.train_schemes
       if schema.get_type() is "DCE"]
   dce_percentage = len(train_set_dce) / (len(self.train_schemes))
   print("Train set with " + str(dce_percentage * 100)
            + " of DCE schemas after oversampling.")

   # Creating the training and testing sets
   self.train_set = [(features(schema), schema.get_type())
       for schema in self.train_schemes]
   self.test_set = [(features(schema), schema.get_type())
       for schema in self.test_schemes]
   print("Feature set created in " + str(int((time.time() - debut) / 60) + 1) + "
minute(s).")
   debut = time.time()

   # Training the classifier
   self.classifier = self.classifiers[classifier_type].train(self.train_set)
   print("Classifier trained in " + str(int((time.time() - debut) / 60) + 1)
               + " minute(s).")
   debut = time.time()

   # Testing the classifier accuracy
   self.accuracy = nltk.classify.accuracy(self.classifier, self.test_set)
```

```
    print("Accuracy of answers: {} %".format(self.accuracy * 100))
    print("Accuracy computed in " + str(int((time.time() - debut) / 60) + 1)
                   + " minute(s).")
    debut = time.time()

    # Creating the confusion matrix
    self.create_confusion_matrix()
    print("Confusion matrix created in " + str(int((time.time() - debut) / 60) +
1)
                   + " minute(s).")
```

3. Results of classifiers

A lot of classifiers have been generated based on the file: *test_generate_dce_classifier.py*. Many variation of the feature implementation and oversampling have been tested. Various types of results were obtained. Some were showing the class imbalance problem, but other were really satisfying. The following shows some of the results obtained:



```
Accuracy of answers: 80.88235294117648 %
Accuracy computed in 1 minute(s).
        |      u        |
        |      n        |
        |      k        |
        |      n        |
        |      o      D |
        |      w      C |
        |      n      E |
--------+---------------+
unknown | <78.4%> 21.6% |
    DCE |      .    <.>|
--------+---------------+
(row = reference; col = test)

Confusion matrix created in 8 minute(s).
16 minutes to generate the naive bayes.

Analysis of the confusion matrix

Analysis of the most important features
Most Informative Features
           logical_link = 'but'               DCE : unknow =      9.9 : 1.0
           logical_link = 'because'           DCE : unknow =      6.4 : 1.0
           logical_link = ''               unknow : DCE    =      5.7 : 1.0
           snippet_verb = 'S'                 DCE : unknow =      3.9 : 1.0
                snippet = "['PP', 'VBZ', 'RB', 'JJ', 'SENT']" unknow : DCE    =      2.2 : 1.0
                snippet = "['PP', 'VBD', 'JJ', 'SENT']"    DCE : unknow =      2.2 : 1.0
                snippet = "['PP', 'VBD', 'RB', 'VVN', 'SENT']"    DCE : unknow =      2.2 : 1.0
                snippet = "['PP', 'VVD', 'SENT']"    DCE : unknow =      2.2 : 1.0
               sentence = "['NP', 'VBD', 'NP', 'IN', 'NP', 'CC', 'NP', 'VBD', '.']"    DCE : unknow =      2.0 : 1.0
           snippet_verb = 'A'              unknow : DCE    =      1.8 : 1.0
```

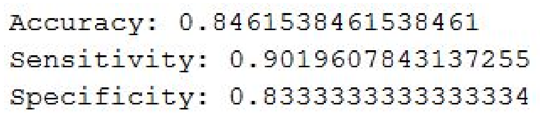
Accuracy = (TP + TN) / (TP + TN + FP + FN) = (16 + 84) / (14 + 102 + 18+ 8) = 0.80
Sensitivity = TP / (TP + FN) = 16 / (16 + 8) = 0.66
Specificity = TN / (TN +FP) = 94 / (94 + 18) = 0.83

This classifier shows interesting results. Even if its sensitivity is not good enough to satisfy the initial objectives, its accuracy is quite good and the list of its most important features is consistent with the expectations discussed in the feature selection.

The best classifier obtains the following results on its testing set:

Accuracy: (TP + TN) / (TP + TN + FP +FN) = (19 + 97) / (19 + 97 + 3 + 17) = 0.852
Sensitivity: TP / (TP + FN) = 19 / (19 + 3) = 0.863
Specificity: TN / (TN + FP) = 97 / (97 + 17) = 0.850

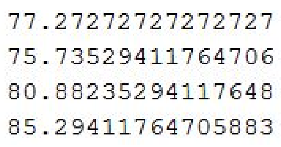
When applied on the full data set, it has these results:



```
Accuracy: 0.8461538461538461
Sensitivity: 0.9019607843137255
Specificity: 0.8333333333333334
```

4. Improving accuracy by bagging

In order to improve the classification, bagging has been used in the file *dce_bagging.py.* Bagging is an ensemble learning method that relies on the vote of many classifiers. The four best classifiers have been used. Their accuracy results were as follow:



```
77.27272727272727
75.73529411764706
80.88235294117648
85.29411764705883
```

Bagging of these four classifiers:

Indicator	***Objective***	***Result***	***Percentage of the objective***
Accuracy	0.9	**0.809**	**89 %**
Sensitivity	0.9	**0.901**	**100 %**
Specificity	0.8	**0.788**	**97 %**

Bagging with more weight given to the best classifier *(its vote counts twice)*.

Indicator	***Objective***	***Result***	***Percentage of the objective***
Accuracy	0.9	**0.915**	**101 %**
Sensitivity	0.9	**0.862**	**95 %**
Specificity	0.8	**0.927**	**116 %**

Even, if the initial objectives have not been completely achieved, applying an ensemble learning method on my classifiers is leading to better classification results and to satisfying results. For the final implementation of the project, the second bagging has been chosen for DCE classification.

Conclusion of the classification:

Detecting different types of Winograd Schemas can be defined as a multi-class supervised classification problem where many class specific binary classifiers can be used together. After discussing our initial objectives and the metrics to measure our achievements, probabilistic rules have been defined to determine if features extracted from the Schema can be good or not. We have applied these rules on a set of features for DCE schemas and have used these features in Naive Bayes classifiers. Ensemble learning has then been used to improve the results. We obtained results which are really close our initial objectives. Some further progress can still be achieved by improving the natural language processing, changing the naive Bayes classifier to a bayesian belief network and by using other ensemble learning techniques as boosting. This approach seems then to be promising.

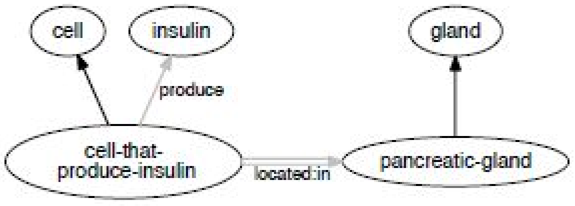
III. Resolution of DCE Winograd Schemas

We have seen that Winograd Schemas can be transformed in textual entailment problems. Especially we focused on DCE schemas which are textual entailment problems based on causal inferences. This part discusses Natural Logic as a solution idea for Winograd Schemas.

A. Natural logic and graph representation

Natural logic - Natural logic is a system of logical inference which operates over nature language [10] and which has been developed to work on textual inference and entailment. Natural Logic has been developped in order to overcome the translation problem of natural language to First Order Logic. Whereas FOL is known as really effective and precise for inference deduction, it is really hard to transform automatically natural language into first order logic. As a difference to FOL, natural language emphasis more on the central role of the verb in natural language, especially in affirmative sentences.

Graphical representation - Natural logic is also introducing specific graph forms named concept graphs in which query between terms can be conducted. All sentences gathered in the database are separated in different concepts in the graph which representation. Each concept is supposed to be unique and shared across the whole database. Concept graphs are introducing among others two types of relations: class inclusion and restrictions. In the paper [11], a simple, but efficient example is given:



In this example, generalization relations are represented in black as between "cell-that-produce-insulin" and "cell", restriction relations are represented in grey. Single arrows are representing elements related to the concept definition, whereas double arrows are not.

One of the main advantages of this notation is the ability to perform graph queries to determine inferences based on the concept relations.

B. Solving DCE schema as natural logic entailment

1. Resolution method

Inspired from the natural language graphical representation, this part tries to establish a resolution method for Winograd Schemas. It does not target to use perfectly natural language notations, but more to explain how this could be used in the context of Winograd Challenge.

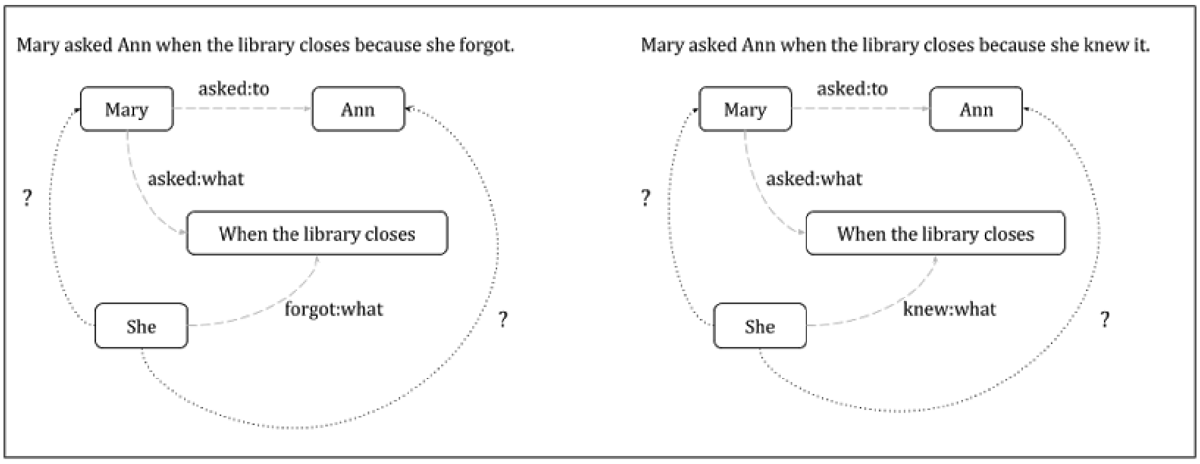
Let's consider the example:

Mary asked Ann when the library closes because she ***(forgot / knew)*** *it.*
Who ***(forgot / knew)*** *it?*

We can transform this schema into textual entailment problems to obtain:

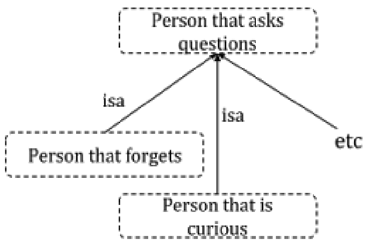
T: *Mary asked Ann when the library closes because she forgot it.*
H1: *Mary forgot it.*
H2: *Ann forgot it.*

Similarly, we obtain the second entailment recognition problem by replacing *forgot* by *knew*. Based on the main idea of natural language representation, entailment problems (and thus, the Winograd Schema) we can be represented as follows:

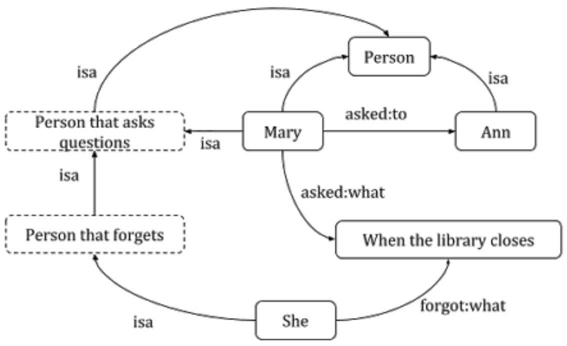


Grey color has been used for restriction relation and dotted arrows to represent double arrows. The keyword **isa** is used to represent concept inclusion.

Based on natural language processing some information can be added on this representation. It can be determined that *Mary* and *Ann* are included in *person* and that *she* is related to the concept of *person-that-forgets*. Common knowledge from other graphs in the database should then be used to find the missing piece of knowledge. The needed information is here: *Person that forgets* is a *person that asks question.*

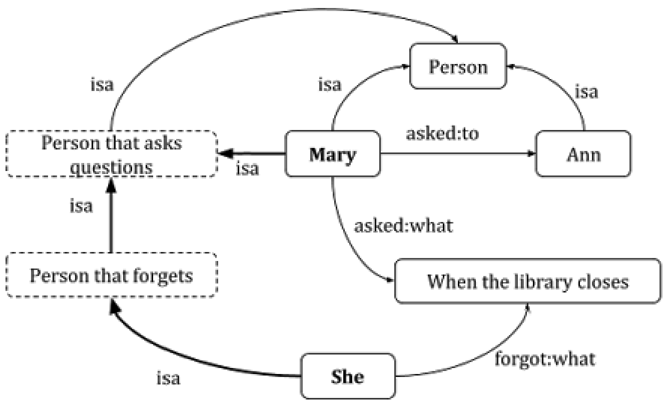


The following representation is then obtained:



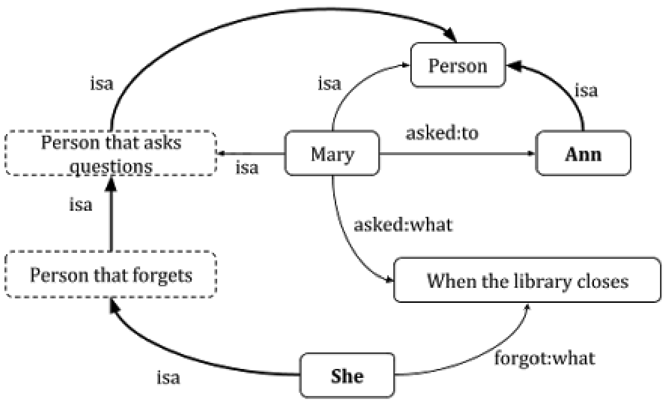
In order to determine the reference of the pronoun *she,* two queries are conducted in order to construct the shortest path based on **isa** relations from the pronoun *she* to each solution excluding the other other solution.

The two following paths are then constructed:



<u>Path from *She* to *Mary* excluding *Ann:*</u>

She which is a *person that-forgets* which is a *person-that-asks-questions* whereof is *Mary*.

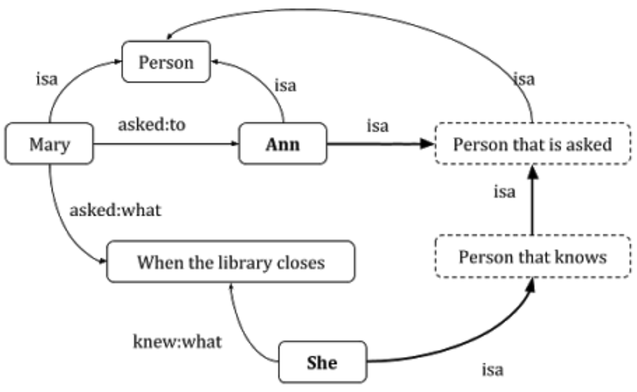


<u>Path from *She* to *Ann* excluding *Mary:*</u>

She which is a *person-that-forgets* which is a *person-that-asks-questions* which is a *person* whereof is *Ann*.

The path from *She* to *Mary* is shorter and then more likely to be the correct answer. An hint in order to eliminate Ann as the correct answer is that the path from *she* to *Ann* is going through the relation *Person* which can equally refer to both answers.

Similarly, for the other part of the Winograd Schema, by adding the common piece of knowledge *person that are asked* is a *person that knows*:



<u>Path from *She* to *Ann* excluding *Mary*:</u>

She which is a person that knows which is a person that is asked questions whereof is Ann.

<u>Path from *She* to *Mary* excluding *Ann*:</u>

She which is a person that knows which is a person that is asked questions which are persons whereof is Mary.

The shortest path is now from She to Ann. Ann is then selected as the answer.
Based on natural logic representation, the underlying textual inference problem of the Winograd Schema has been successfully solved as follows:

- Mary asked Ann when the library clauses because she forgot. → Mary
- Mary asked Ann when the library clauses because she knew it. → Ann

2. Problem simplification and resolution rule

With this approach, the main difficulty of the Winograd Challenge would then be to create a knowledge database containing enough information in order to solve a wide range of Winograd Schemas. Due its difficulty, creating my own knowledge database cannot be considered as a solution. A simplification is then proposed.

This simplification is based on the statement that the length of the two paths obtained during the resolution reflects the similarity between the related concepts. Since these paths always involves the main elements of the basis of the sentence and of the snippet, analysing their similarity could help to get the answer of the schema.

Other elements also have to be considered like the type of logical link used in the DCE schema or the potential negation of the main information.

DCE Winograd schema (*X action1 Y LL Z action2)* can be resolved as:

- *If LL is causal link:*
    - *If action1 and action2 are similar then X* (C1)
    - *If action1 and opposite of action2 are similar then Y* (C2)
    - *Else unknown* (C3)
- *If LL is opposition link:*
    - *If action1 and action2 are similar then Y* (O1)
    - *If action1 and opposite of action2 are similar then X* (O2)
    - *Else unknown* (O3)

This resolution method has been tried by hand on DCE schemas and has shown promising results. The following shows an example covering both negation and causality relations:

The forward player ***(scored / missed)*** *against the goalkeeper*
(because / although) *he was very (****good / bad****).*
Who was very ***(good / bad)*** *?*

1. The forward player **scored** to the goalkeeper **because** he was very **(good / bad)**.
   Who was good (bad)?

The forward player scored against the goalkeeper because he was very good. The relation is causal, to score is some kind of success which is related to good. We have case **C1** and the answer is then X, the **forward player**.

The forward player scored against the goalkeeper because he was very bad. The relation is causal, to score is some kind of success which is not related to bad. However, the opposite of bad is good which is related to score. We have case **C2** and answer is then Y, the **goalkeeper**.

2. The forward player **missed** against the goalkeeper **because** he was very **(good / bad)**.
   Who was good (bad)?

The forward player missed against the goalkeeper because he was very good. The relation is causal, to score is some kind of failure which is not related to good. However, the opposite of good is bad which is related to failure. We have case C2 and answer is then Y, the goalkeeper.

The forward player missed against the goalkeeper because he was very bad. The relation is causal, to score is some kind of failure which is not related to bad. We have case **C1** and the answer is then Y, the **goalkeeper**.

3. The forward player **scored** against the goalkeeper **although** he was very good (bad). Who was good (bad)?

The forward player scored against the goalkeeper although he was very good. The relation is a concession (opposition), to score is some kind of success which is related to good. We have case **O1** and the answer is then Y, the **goalkeeper**.

The forward player scored against the goalkeeper although he was very bad. The relation is a concession (opposition), to score is some kind of success which is not related to bad. However, the opposite of bad is good which is related to score. We have case **O2** and answer is then X, the **forward player**.

4. The forward player **missed** against the goalkeeper **although** he was very good (bad). Who was good (bad)?

The forward player missed against the goalkeeper although he was very good. The relation is a concession (opposition), miss is some kind of failure which is not related to good. However, the opposite of good is bad which is related to failure. We have case **O2** and answer is then X, the **forward player.**

The forward player missed against the goalkeeper although he was very bad. The relation is concession (opposition), miss is some kind of failure which is not related to bad. We have case **O1** and the answer is then Y, the **goalkeeper**.

Results - We have designed here a solution that is able to work also when a negation is used or when the logical link is changing. It seems to be working well when the similarity relationships look obvious. We will need to discuss later the results obtained in more tricky cases, but it looks promising.

C. Implementation of the resolution algorithm

Resolution algorithm is made in the file *dce_solver.py.* The first part of the resolution algorithm is based on a method extracting the features needed for the resolution from the schema.

```
def features(schema):

    feature_set = {}

    # Getting the type of relation
    feature_set['causal_relation'] = is_causal_relation(schema)
    feature_set['opposition_relation'] = is_opposition_relation(schema)

    # Getting the actions from both sentences
    action1 = get_action(schema)
    trait = get_trait(schema)
    antonym_trait = antonym(trait)

    try:
        feature_set['action_trait_similarity'] =
                        similarity(action1, trait)
    except Exception as e:
        feature_set['action_trait_similarity'] = -1
    try:
        feature_set['action_!trait_similarity'] =
                        similarity(action1, antonym_trait[0])
    except Exception as e:
        feature_set['action_!trait_similarity'] = -1

    # Features not used, but saved for result discussion
    feature_set['action'] = action1
    feature_set['trait'] = trait
    feature_set['antonym'] = antonym_trait
    feature_set['action_snippet'] = get_snippet_verb(schema)

    return feature_set
```

Some of the features used in the resolution have already been described in the classification part as: *is_causal_relation* and *is_opposition_relation*.

The functions *get_action* and *get_trait* are functions returning the main element of the basis of the sentence and the main element of the snippet. As stated by natural logic, it is assumed here that most of the meaning of a sentence is contained in the verb. In the implementation, the main element is then the verb or the following adjectives when the verb does not give any information. Returning this meaningful element is done as follows:

```
def get_main_element(sentence):
    # If verb is "be" then return the next JJ.
    tokens = analyze(sentence)

    for i in range(0, len(tokens), 1):
        if "VV" in tokens[i].postag:
            return tokens[i].lemma
        else:
            # Verb be
            if "VB" in tokens[i].postag or "VH" in tokens[i].postag:
                if tokens[i + 1].postag == "VVG":  # verb in be+ING
                    return tokens[i + 1].lemma
                if tokens[i + 1].postag == "RBR":  # if comparative
                    return tokens[i + 1].lemma
                if tokens[i + 1].postag == "RBS":  # if superlative
                    return tokens[i + 1].lemma
                for j in range(i, len(tokens), 1):
                    if tokens[j].postag in ["JJ", "JJS", "JJR"]:
                        return tokens[j].lemma
                    if tokens[j].postag == "NN":
                        return tokens[j].lemma
```

After tokenizing and parsing the sentence, the algorithm focuses on the verb "be". This treatment should most probably be extended to other specific verbs. However, this function still shows interesting results. Its accuracy will be discussed in detail later on.

The similarities between concepts are then computed using the project ConceptNet [12].

```
def similarity(word1, word2):
    """
    Use concept net in order to find the similarity between two words.
    The order of the input doesn't not have any impact.
    Request are made using the urllib on ConceptNet website.

    :return: similarity coefficient as float between 0 and 1.
    """

    def sim(w1, w2):
        link = "http://conceptnet5.media.mit.edu/data/5.4/assoc/c/en/"
               + w1 + "?filter=/c/en/" + w2 + "/.&limit=1"
        response = urlopen(link).read().decode('utf8')
        obj = json.loads(response)
        try:
            return obj["similar"][0][1]
        except IndexError:
            return 0

    return float(sim(word1, word2) + sim(word2, word1)) /2
```

The main part of the resolution algorithm is made with a tree structure based on *if* conditions.

```
def solve(schema):

   if schema.get_type() == "DCE":

       feature_set = features(schema)

       simi = feature_set['action_trait_similarity']
       dissimi = feature_set['action_!trait_similarity']

       if feature_set['causal_relation']:
           # if action and !trait are more related, then second element
           if simi > 0 and simi > dissimi:
               return schema.answer_A, ((simi - dissimi) / simi)

           # if action and !trait are more related, then first element
           if dissimi > 0 and dissimi > simi:
               return schema.answer_B, ((dissimi - simi) / dissimi)

           return None

       if feature_set['opposition_relation']:
           # if action and !trait are more related, then second element
           if simi > 0 and simi > dissimi:
               return schema.answer_B, ((simi - dissimi) / simi)

           # if action and !trait are more related, then first element
           if dissimi > 0 and dissimi > simi:
               return schema.answer_A, ((dissimi - simi) / dissimi)

           return None
       else:
           return None
   else:
       return None
```

D. Result discussion

1. Simple example

Let's use a new example and see the features applied to it:

Metz FC won against the one from Paris because it was ***(better / bad)****.*
Who was ***(better /bad)****?*

The algorithm returns the following features and results for the two sentences:

1) ***Metz FC won against the one from Paris because it was better.***

```
---- ANSWERING THE FIRST SCHEMA ----
Metz football team won against the one from Paris because it was better.
Action: win - Trait: well - !Trait: ['ill']
action_!trait_similarity -> 0.0
causal_relation -> True
opposition_relation -> False
action_trait_similarity -> 0.007171918575410l095

Answer: ('Metz football team', 1.0)
```

2) ***Metz FC won against the one from Paris because it was bad.***

```
---- ANSWERING THE SECOND SCHEMA ----
Metz football team won against the one from Paris because it was bad.
Action: win - Trait: bad - !Trait: ['unregretful', 'good']
action_!trait_similarity -> 0.0029181530379887876
causal_relation -> True
opposition_relation -> False
action_trait_similarity -> 0.0

Answer: ('the one from Paris', 1.0)
```

Both examples are then giving satisfying results. Confidence has been defined as the maximum of the similarities coefficients minus their minimum divided by their maximum.

2. Testing robustness of the algorithm on a simple example

*The forward player **(scored / missed)** against the goalkeeper*
***(because / although)** he was very (**good / bad**).*
*Who was very **(good / bad)** ?*

Case	***Logical link***	***Action***	***Snippet***		***Expected***	***Predicted***	***Correct?***	***Confidence***
1	because (causal)	scored	good		F. Player	F. Player	**Yes**	1
			bad		Goalkeeper	Goalkeeper	**Yes**	0.785
2		missed	good		Goalkeeper	Goalkeeper	**Yes**	1
			bad		F. Player	F. Player	**Yes**	1
3	although (opposition)	scored	good		Goalkeeper	Goalkeeper	**Yes**	0.785
			bad		F. Player	F. Player	**Yes**	0.785
4		missed	good		F. Player	F. Player	**Yes**	1
			bad		Goalkeeper	F. Player	**No**	0.594

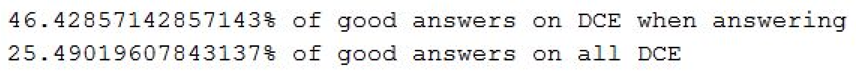
On these example we can see that the accurateness is quite good. Even if one error has been performed, its confidence value is lower than the one obtained for correct answers which could be a way to trigger it in advance.

3. Applying on the full data set

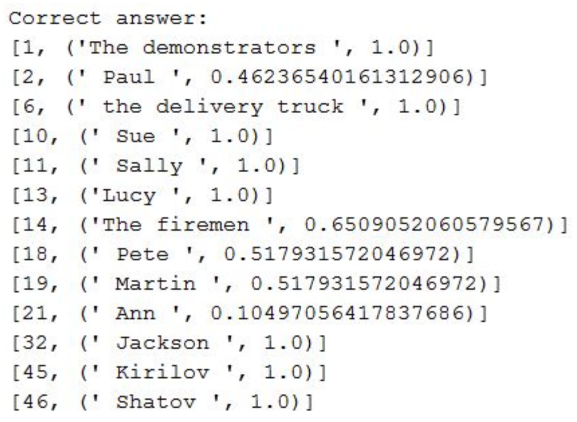
In order to test how our solution is performing, the resolution process has been used on all the DCE schema identified in the xml file. The following code has been used:

```
def test_solver(self):
    schemes = parse_xml()
    add_labels(schemes)
    schemes = [schema for schema in schemes if schema.get_type() == "DCE"]
    print(len(schemes))
    right, wrong, silent = [], [], []

    for schema in schemes:
        guess = self.resolve(schema)
        if schema.correct == guess[0]:
            right.append([schemes.index(schema), guess])
        elif 'unable' in guess:
            silent.append([schemes.index(schema), guess])
        else:
            wrong.append([schemes.index(schema), guess])

    answers = len(wrong) + len(right)
    print(str(len(right) / answers * 100)
            + "% of good answers on DCE when answering")
    print(str(len(right) / len(schemes) * 100)
            + "% of good answers on all DCE")

    print("Correct answer:")
    [print(r) for r in right]
    print("Wrong answer:")
    [print(r) for r in wrong]
    print("No answer:")
    [print(r) for r in silent]

@staticmethod
def resolve(schema):
    dce_solver = DirectCausalEventSolver()
    answer = dce_solver.solve(schema) if dce_solver.solve(schema) is not
None else "unable to predict"
    return answer
```

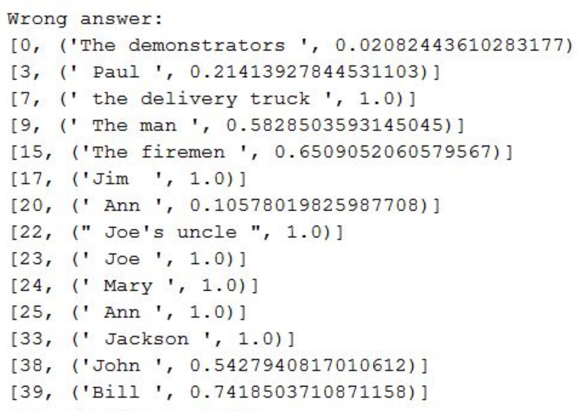
The code is then returning many interesting information such as the percentage of correct answers on all the DCE schemas and the percentage of correct answers when the solver has proposed a solution.



```
46.42857142857143% of good answers on DCE when answering
25.49019607843137% of good answers on all DCE
```



```
Correct answer:
[1, ('The demonstrators ', 1.0)]
[2, (' Paul ', 0.4623654016131290 6)]
[6, (' the delivery truck ', 1.0)]
[10, (' Sue ', 1.0)]
[11, (' Sally ', 1.0)]
[13, ('Lucy ', 1.0)]
[14, ('The firemen ', 0.6509052060579567)]
[18, (' Pete ', 0.517931572046972)]
[19, (' Martin ', 0.517931572046972)]
[21, (' Ann ', 0.10497056417837686)]
[32, (' Jackson ', 1.0)]
[45, (' Kirilov ', 1.0)]
[46, (' Shatov ', 1.0)]
```



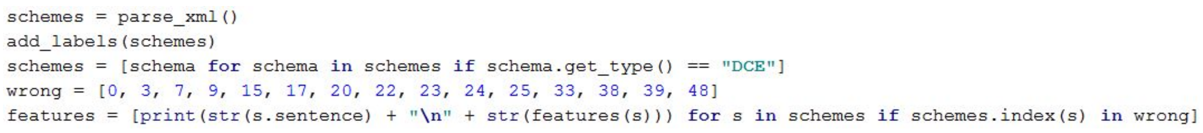
```
Wrong answer:
[0, ('The demonstrators ', 0.02082443610283177)
[3, (' Paul ', 0.21413927844531103)]
[7, (' the delivery truck ', 1.0)]
[9, (' The man ', 0.5828503593145045)]
[15, ('The firemen ', 0.6509052060579567)]
[17, ('Jim  ', 1.0)]
[20, (' Ann ', 0.10578019825987708)]
[22, (" Joe's uncle ", 1.0)]
[23, (' Joe ', 1.0)]
[24, (' Mary ', 1.0)]
[25, (' Ann ', 1.0)]
[33, (' Jackson ', 1.0)]
[38, ('John ', 0.5427940817010612)]
[39, ('Bill ', 0.7418503710871158)]
```

Note: *the index displayed here are the index among the DCE elements, not among all schemas.*

The solver demonstrates a score that is relatively low, 46%, when trying to answer questions The confidence indicators seems also not to be trustworthy. This can be explained.

NLP failures - First of all, some of the errors come from the natural language processing applied here. Especially the one used in the search for an antonym which should be improved. In fact, we should look for an antonym that perfectly matches the context of the sentence which is also not easy to find. A lot of mistakes are introduced here because the antonym used for the similarity searches are not the right ones, sometimes finding the right antonym also implies to be more abstract.

Let's analyse the features of the 14 wrong cases:



```
schemes = parse_xml()
add_labels(schemes)
schemes = [schema for schema in schemes if schema.get_type() == "DCE"]
wrong = [0, 3, 7, 9, 15, 17, 20, 22, 23, 24, 25, 33, 38, 39, 48]
features = [print(str(s.sentence) + "\n" + str(features(s))) for s in schemes if schemes.index(s) in wrong]
```

ID	***Link (expected)***	***Action (expected)***	***Trait (expected)***	***Antonym (expected)***	***Confidence***
0	causal	refuse	fear	encouragement	0.02
3	opposition	**try (call)**	available	unavailable	0.21
7	causal	**Zoom (zoom by)**	**Go (slow)**	**No-go**	1.0
9	causal	lift	heavy	light	0.58
15	causal	arrive	**come (far away)**	**Disappear**	0.65
17	causal	comfort	upset	**methodical**	1.0
20	causal	know	nosy	indifferent	0.10
22	opposition	beat	young	old	1.0

23	opposition	beat	old	young	1.0
24	causal	ask	forget	**recollect (remember)**	1.0
25	opposition	ask	forget	**recollect (remember)**	1.0
33	opposition	influence	live	**recorded (dead)**	1.0
38	causal	pass	full	**empty (hungry)**	0.54
39	causal	pass	hungry	**thursty (full)**	0.74

In this tables, unexpected values are in bold. 9 out of 14 errors are due to natural language processing errors.

- Some are related to tricky cases of detection of the main action. In sentence number 3, *try* that is detected instead of *call* in *try to call*. In index 7, the verb *zoom by* is not detected as a phrasal verbs and only *zoom* is returned.
- Index number 7 and 15 are related to errors in the detection of the meaningful part of the snippet.
- Index 15,17, 24, 25, 33, 38 and 39 are related to problems while choosing the antonym. The one chosen is not the best one reflecting the situation and the similarity computed with the action is then also wrong in the context of the sentence.

Most of these errors are related to high confidence. As a conclusion, by improving the features, especially *get_main_element,* the number of correct answers could increase significantly.

Inherent problem - The other important element is that the algorithm is underestimating the complexity of the scheme resolution. This resolution process assumes that elements used for resolution are directly related which is not always the case.

Winograd Schema n°3-4:
The trophy would not fit in the brown suitcase because it was too **(big / small)**.
What was too **(big / small)**?

When resolving this schema, the solver is not able to predict an answer because it cannot find a direct relation between the elements *big / small* and *fit.* Human mind would detect that a suitcase in a container and would focus on the relation between container and *big / small.* Solving this schema involves then some extra steps that the algorithm cannot handle.

This schema is both using causal and spatial inference. It raises then here the question of the DCE schema class as being too wide. In fact, a simple solution for this schema would be to create a classifier specific to "*container problem*". Then, instead of comparing *big* with *fit,* a resolution algorithm comparing the word *container* with *big* or *small* could be implemented.

However, we can note that our resolution is then able to perform correctly on basic cases where the answers possessed an attribute or a quality (similar to Causal Attributes Schemas discussed in the first part) such as the following ones:

Winograd Schema n°23-24:
Although they ran at about the same speed, Sue beat Sally because she had such a **(good / bad)** start.
Who had a **(good / bad)** start?

Or,

Winograd Schema n°39-40:
Pete envies Martin **(although / because)** he is very successful.
Who is very successful?

Or,

Winograd Schema n°262-263:
Kirilov ceded the presidency to Shatov because he was **(more / less)** popular.
Who was **(more / less)** popular?

Conclusion of the resolution:

In order to solve Winograd Schemas, we took inspiration in natural logic which provides a graphical representation based on concepts. This representation allows to perform queries to determine inferences. However, applying natural logic based solutions involve to have a preformatted knowledge base that is large enough. In order to work around the problem, a solution has been proposed based on similarity relations between the keywords of the schema and their antonyms. This solution seems to be working on simple cases among the Causal Attributives (subpart of DCE), but does not seem to be sufficient on DCE. Even if some progress could be achieved by improving the natural language processing in the feature extraction of the schemas, this solution seems to underestimate the complexity of the problem. A solution would then be to divide DCE in many subclasses and apply specific variation of this algorithm on each of them.

Conclusion:

The aim of this project was to analyse Winograd Schemas and to propose a solution in order to solve (at least) some of them. Winograd Schemas are based on two sentences containing a pronoun whose reference is ambiguous and can only be determined based on common knowledge. By transforming Winograd Schema into textual entailment problems, we have been able to highlight the main ideas of a classification based on their underlying inference types. The case of Direct Causal Events, schemas based on causal inference, has been chosen to discuss classification and resolution of Winograd Schemas. Detecting different types of Winograd Schemas is as a multi-class classification problem. A possible solution is to use various class specific binary classifiers together. Based on some probabilistic rules for extraction of features inside schemas, a set of features for DCE has been created. Using different methods such as bagging, we have been able to get to an accuracy of 91% and the classifier is currently able to detect 86% of DCE schemas correctly. Some further improvement ideas have been discussed in the report. A solution for DCE schemas has then been discussed based on the graphical concept representation of natural logic. In order to work around the problem of creating a conceptual database, a simplification rule based on keyword similarity has been introduced and tested. Even if this simplification shows some limitations, by underestimating the complexity of the resolution process, it seems able to solve one type of DCE schemas defined as Clausal Attributive Schemas. A solution seems then to keep dividing the DCE schemas in subclasses in order to apply specific variations of the resolution algorithm on each of them.

Appendix

A. References

List of the references directly used inside of the report:

[1] Wikipedia Article on the Turing Test, https://en.wikipedia.org/wiki/Turing_test

[2] What is the Winograd Schema Challenge? http://commonsensereasoning.org/winograd.html

[3] Collection of Winograd Schemas proposed by the New York University, http://www.cs.nyu.edu/faculty/davise/papers/WinogradSchemas/WSCollection.xml

[4] Steven Bird, Erwan Klein & Edward Loper, *Natural Language Processing with Python*, Chapter 6: Learning to classify texts, http://www.nltk.org/book/ch06.html

[5] Arpit Sharma, Nguyen H. Vo, Shruti Gaur & Chitta Baral, *An Approach to Solve Winograd Schema Challenge Using Automatically Extracted Commonsense Knowledge*.

[6] Treetagger official webpage, http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/

[7] List of tags of TreeTagger, http://courses.washington.edu/hypertxt/csar-v02/penntable.html

[8] White Smoke, *Stative verbs*, http://www.whitesmoke.com/stative-verbs

[9] Machine Learning Mastery, *8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset,* http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

[10] Bill MacCarntey, Christopher D. Manning, *Natural Logic for Textual Inference,* http://nlp.stanford.edu/~wcmac/papers/natlog-wtep07.pdf

[11] Troels Andreasen ,Henrik Bulskov, Jørgen Fischer Nilsson & Per Anker Jensen, *A System for Conceptual Pathway Finding and Deductive Querying,* http://link.springer.com/chapter/10.1007%2F978-3-319-08326-1_27

[12] ConceptNet main webpage, http://conceptnet5.media.mit.edu/

Additional references used during the project:

- Jørgen Fischer Nilsson, *In Pursuit of Natural Logics for Ontology-Structured Knowledge Bases*
  https://www.thinkmind.org/index.php?view=article&articleid=cognitive_2015_3_20_40044

- Interview of Charlie Ortiz by Evan Ackerman, *Winograd Schema Challenge Results: AI Common Sense Still a Problem, for Now*
  http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/winograd-schema-challenge-results-ai-common-sense-still-a-problem-for-now

- Eclectic English, *English Action and State verbs,*
  http://www.eclecticenglish.com/grammar/PresentContinuous1H.html

- www.perfect-english-grammar.com, *Stative verbs list*
  http://www.perfect-english-grammar.com/support-files/stative-verbs-list.pdf

- Wikipedia Article on Common Sense, https://en.wikipedia.org/wiki/Common_sense

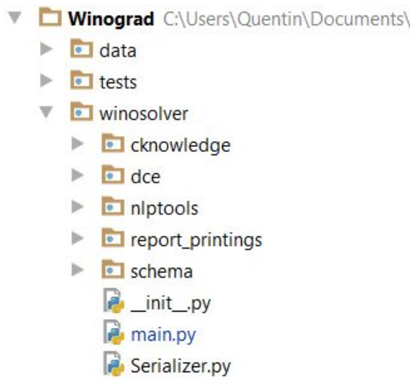
- Wikipedia Article on Common Knowledge,
  https://en.wikipedia.org/wiki/Common_knowledge

- Wikipedia Article on Anomaly detection
  https://en.wikipedia.org/wiki/Anomaly_detection

- Quora discussion on Inference and Entailment,
  https://www.quora.com/What-is-the-difference-between-inference-and-entailment

B. Structure of the submitted project

The project available on the github repository is composed of 3 main folders, *data, tests* and *winosolver.*



```
Winograd C:\Users\Quentin\Documents\
    data
    tests
    winosolver
        cknowledge
        dce
        nlptools
        report_printings
        schema
        __init__.py
        main.py
        Serializer.py
```

The folder *data* is a folder containing the serialized classifiers obtained during the project. It also contains a sentence database from Wikipedia (see Appendix C).

The folder *tests* contains all units tests implemented during the project development. It also contains the scripts used to generate the classifiers.

The folder *winosolver* contains the source code of the project and is performing all the treatment process to classify and solve schemas.

However, the project does not need all classes to be run. The following list of classes represents the ones necessary for DCE classification and resolution. These are the classes that have been discussed in the project.

main.py
Serializer.py
dce/dce_bagging.py
dce/dce_classifier.py
dce/dce_solver.py
dce/features_tools.py
nlptools/Chunker.py
nlptools/GrammaticalClassification.py
nlptools/Tokenizer.py
schema/Schema.py
schema/WSCollection.py
schema/XMLParser.py

Minimal requirements to run the project:
The project directly uses the packages: *nltk, pydictionary, treetaggerwrapper* and *untangle.*

In order to run the project, the packages listed in *requirements.txt* should be installed.

PyDictionary==1.5.2	→ Library used to get antonym of words
beautifulsoup4==4.5.1	→ *dependency library*
click==6.6	→ *dependency library*
futures==3.0.5	→ *dependency library*
goslate==1.5.1	→ *dependency library*
nltk==3.2.1	→ Library for Natural Language Processing in Python
requests==2.11.1	→ *dependency library*
six==1.10.0	→ Library fixing compatibility issues between Python 2 and Python 3
treetaggerwrapper==2.2.4	→ Library used to part to speech tagging (refers to TreeTagger)
untangle==1.1.0	→ Library for XML parsing

C. Resolution of the 2 schemas preloaded in main.py

The following shows the console output of the file *main.py* which can be used to run two preloaded examples from the console. This code can be run based on the submitted project.

```
C:\Users\Quentin\Documents\GitHub\Wino2.0\winovenv\Scripts\python.exe
C:/Users/Quentin/Documents/GitHub/Winograd/winosolver/main.py
Run preloaded examples? (y/n) y

Resolution of the first example:
----- Winograd Schema number 1 -----
Text: Metz football team won against the one from Paris because it was better.
Snippet: it was better
Choices for 'it': A) Metz football team,  or B) the one from Paris
Answer: unknown
Source: console

----- Winograd Schema number 1 -----
Text: Metz football team won against the one from Paris because it was bad.
Snippet: it was bad
Choices for 'it': A) Metz football team,  or B) the one from Paris
Answer: unknown
Source: console

---- CLASSIFYING THE SCHEMA ----
Metz football team won against the one from Paris because it was better.
-> Classified as DCE
Metz football team won against the one from Paris because it was bad.
-> Classified as DCE

---- ANSWERING THE SCHEMA ----
Metz football team won against the one from Paris because it was better.
-> Answer: ('Metz football team', 1.0)
Metz football team won against the one from Paris because it was bad.
-> Answer: ('the one from Paris', 1.0)

Resolution of the second example:
----- Winograd Schema number 38 -----
Text: Pete envies Martin although he is very successful.
Snippet: he  is very successful.
Choices for ' he ': A)  Pete ,  or B)  Martin
Answer:  Pete
Source:  Ernest Davis
```

```
----- Winograd Schema number 39 -----
Text: Pete envies Martin because he  is very successful.

Snippet:  he is very successful.

Choices for ' he ': A)  Pete ,  or B)  Martin
Answer:  Martin
Source:  Ernest Davis

---- CLASSIFYING THE SCHEMA ----
Pete envies Martin although he  is very successful.
-> Classified as DCE

Pete envies Martin because he  is very successful.
-> Classified as DCE

---- ANSWERING THE SCHEMA ----
Pete envies Martin although he  is very successful.
-> Answer: (' Pete ', 0.517931572046972)
Pete envies Martin because he  is very successful.
-> Answer: (' Martin ', 0.517931572046972)

---- END ----
Appuyez sur une touche pour continuer...
```

D. Additional work, finally not used

Some work that is not used has been done in the folder *cknowledge.* This work was intended to be a solution to solve Winograd Schemas based on frequent pattern mining and common knowledge extraction from the internet. Since most of the common knowledge for human is acquired from reading, creating a system able to access a huge database of sentences could be interesting in the context of the Winograd Challenge. I tried to achieve it by two different results.

1. Getting common sense Knowledge from Google Search

In order to get common knowledge, my first approach has been to use Google Custom Search. I created a custom search engine using Google Api to get an API key. I then create a function able to perform requests on my custom search engine in *GoogleSearch.py*:

```
def google_search(self, request, **kwargs):
   self.request = request
   service = build("customsearch", "v1", developerKey=self.api_key)
   res = service.cse().list(q=self.request,
                            cx=self.cse_id,
                             **kwargs).execute()
   if res is None:
       self.gse_results = []
   else:
       # pprint.pprint(res)
       self.gse_results = res['items']
   return self.gse_results
```

The result of this requests is a set of website results containing information such as the website link, its author or the website HTML text. I created then a function able to delete all the HTML characters and useless empty lines into a set of tokenized sentences. The code presented below used to work and had been tested. However it has not been maintained after implementation after the project started to focus on DCE classification and resolution.

```
def tokenize_gse_result(self, gse_result):

   url = gse_result[u'link']

   try:
       # Converting the HTML to bytes
       f = urllib.request.urlopen(url)
       mybytes = f.read()  # read bytes from url
       f.close()
       mystr = mybytes.decode("utf8")   # decodes bytes to string

       # Deleting all the HTML characters
       soup = BeautifulSoup(mystr, "html.parser")
       raw_text = soup.get_text()
```

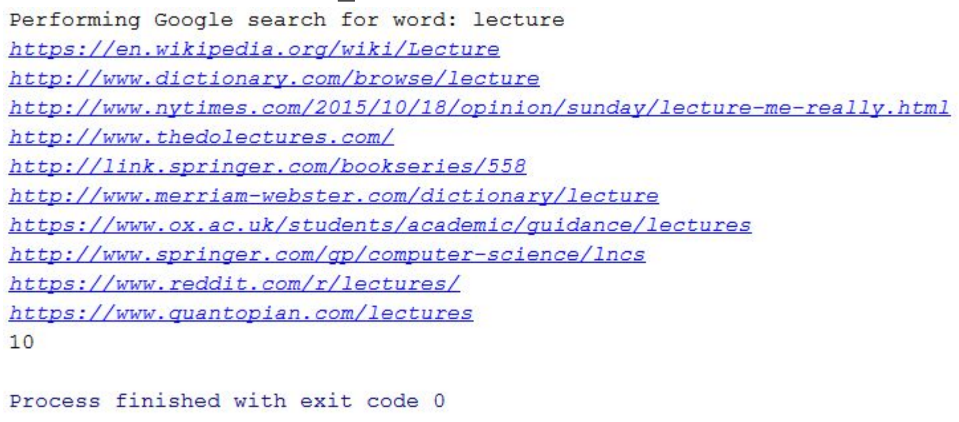
```
        # Deleting empty lines
        raw_text = os.linesep.join([s for s in raw_text.splitlines() if s])

        # Tokenizing the text into sentences based on punctuation
        sentences = self.tokenizer.tokenize_gse_result(raw_text)

        # Tokenizing the text into sentences based on layout
        set = []
        for sentence in sentences:
            lines = sentence.splitlines()
            for line in lines:
                word_groups = line.split('\\s{2,}')
                for group in word_groups:
                    set.append(group)
        return set
    except Exception as e:
        return []
```

All results were returned as follows:

```
def generate_sentences(self, gse_results):
   return [self.tokenize_gse_result(result) for result in gse_results]
```

Example:

```
def test_research(self):
   gs = GoogleSearch()
   request = 'lecture'
   print("Performing Google search for word: " + request)
   gse_pages = gs.google_search(request)
   search_sentences = gs.generate_sentences(gse_pages)

   print(len(search_sentences))
```



```
Performing Google search for word: lecture
https://en.wikipedia.org/wiki/Lecture
http://www.dictionary.com/browse/lecture
http://www.nytimes.com/2015/10/18/opinion/sunday/lecture-me-really.html
http://www.thedolectures.com/
http://link.springer.com/bookseries/558
http://www.merriam-webster.com/dictionary/lecture
https://www.ox.ac.uk/students/academic/guidance/lectures
http://www.springer.com/gp/computer-science/lncs
https://www.reddit.com/r/lectures/
https://www.quantopian.com/lectures
10

Process finished with exit code 0
```

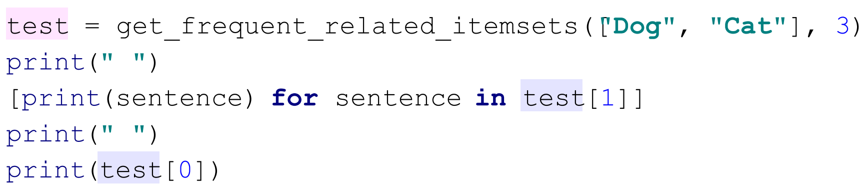
This approach has finally not been used the project because of the pricing limitation of the Google API which only allows a limited number of requests per day.

2. Getting common sense Knowledge from Wikipedia Articles

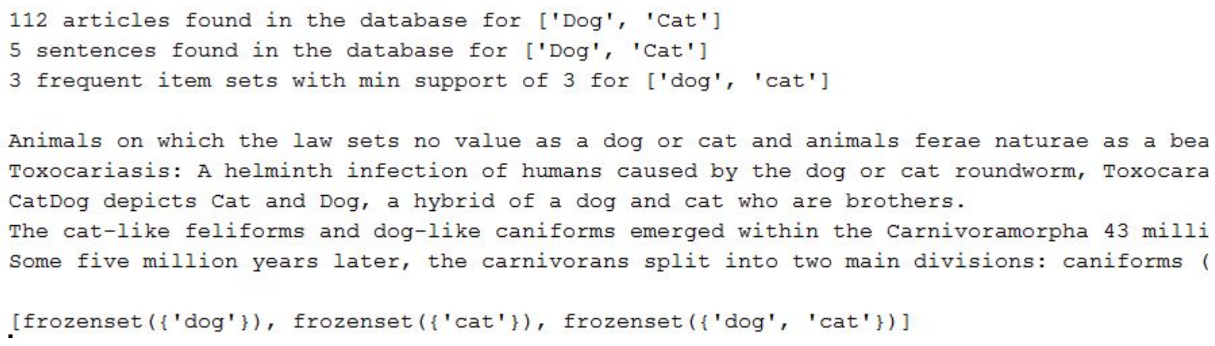
As an alternative to Google Search, I focused on using a library to download wikipedia articles. I created a script able to use the library *wikipedia* for Python, in order to download and tokenize wikipedia articles in the file *CreateWikiDB.py*. The database currently 2376 wikipedia articles, but at least twice more could have been generated.

Based on this database, I was then able to perform searches to get all the sentences where a set of words were used together. Code is available in *SentenceDatabase.py*. Based on the file *structure_mining.py*, I have then been able to get frequent itemsets inside these sentences.

Example:
The following request returns the frequent itemsets in the database with a minimum support of 2 and containing at least one of the words in the research.



```
test = get_frequent_related_itemsets(["Dog", "Cat"], 3)
print(" ")
[print(sentence) for sentence in test[1]]
print(" ")
print(test[0])
```



```
112 articles found in the database for ['Dog', 'Cat']
5 sentences found in the database for ['Dog', 'Cat']
3 frequent item sets with min support of 3 for ['dog', 'cat']

Animals on which the law sets no value as a dog or cat and animals ferae naturae as a bea
Toxocariasis: A helminth infection of humans caused by the dog or cat roundworm, Toxocara
CatDog depicts Cat and Dog, a hybrid of a dog and cat who are brothers.
The cat-like feliforms and dog-like caniforms emerged within the Carnivoramorpha 43 milli
Some five million years later, the carnivorans split into two main divisions: caniforms (

[frozenset({'dog'}), frozenset({'cat'}), frozenset({'dog', 'cat'})]
```

These printings shows that we have been able to find 112 articles containing at least one of the two words. The among these 112 articles, we found 5 sentences that were containing these two words. We found then 3 frequent itemsets.

This example doesn't show any real interest since we don't get any new information, but it shows that the algorithm works. It could be interesting to keep generating the database in order to get more articles (and thus more sentences) to work on.

Minimal requirements to run the project:
All dependencies needed to run all part of the project is defined in *requirements_all.txt.*

```
args==0.1.0
beautifulsoup4==4.5.1
bs4==0.0.1
click==6.6
clint==0.5.1
futures==3.0.5
google-api-python-client==1.5.1
goslate==1.5.1
httplib2==0.9.2
nltk==3.2.1
numpy==1.11.2
oauth2client==3.0.0
pkginfo==1.3.2
pyasn1==0.1.9
pyasn1-modules==0.0.8
PyDictionary==1.5.2
pymining==0.2
requests==2.11.1
requests-toolbelt==0.7.0
rsa==3.4.2
simplejson==3.8.2
six==1.10.0
treetaggerwrapper==2.2.4
untangle==1.1.0
uritemplate==0.6
virtualenv==15.0.3
wikipedia==1.4.0
winosolver==0.0.1
```

E. Feature quality quantification based on independency

Note*: In this part, I tried to discussed rules to characterize dependencies. However, due to a lack of time, I didn't apply these rules in my project. This is why I chose to add this part in the appendix and remove it from the report.*

This part discusses the independency relation between features. It can be especially interesting in the case of Naive Bayes classifiers since they rely on a conditional independency assumption. Independency between features in the set is important in order to avoid overfitting which means avoiding giving too much weights to some characteristics.

Independent features - Let's consider a fixed α belonging to]0:1]. Two rules Ri and Rj are considered as independent for α, if *dependency(Ri, Rj)* $< \alpha$i,j. The minimum value of αi,j satisfying the inequation can be used to quantify the dependency between Ri and Rj.

Independent set of features - Let's consider a set of rules of length n. Let's consider fixed αi,j the dependency between each pair of rules (i, j) from the feature set. The maximum value α of all αi,j can be used to quantify the dependency of the feature set.

To quantify the rule dependencies, I described the rule *dependency* as follow:

$$\textit{dependency(Ri, Rj)} = \max (| P(Ri) - P(Ri \mid Rj) | / P(Ri), | P(Rj) - P(Ri \mid Rj) | / P(Rj))$$

With, Ri, Rj Bernoulli variables representing the respect of rule number i and rule number j in the set of Winograd schemas and P(Ri), P(Rj) both different from 0.

Informal demonstration - In case of independent rules, then the probability of having the rule Ri in the dataset shouldn't be influenced by having the rule Rj. Then, the conditional probability P(Ri | Rj) shouldn't be differing a lot from P(Ri). The dependency measure should be then close to 0. On the other hand, let's assume that Rj and Ri are dependent. Then we have two cases. Either Ri and Rj are positively correlated, either they are negatively correlated. If they are positively correlated, then P(Ri | Rj) - P(Ri) > 0 and the dependency value will increase. On the other side, if they are negatively correlated then P(Ri) - P(Ri | Rj) > 0. The division by P(Ri) is used to normalize the result and have it between 0 and 1.

Properties - The independency is reflexive, symmetric and non-transitive.

Reflexivity: *dependency(Ri, Ri)* = max (0, 0) = 0 < α.

Symmetry: *dependency(Ri, Rj)* = max (| P(Ri) - P(Ri | Rj) | / P(Ri), | P(Rj) - P(Ri | Rj) | / P(Rj))
= max (| P(Rj) - P(Ri | Rj) | / P(Rj), | P(Ri) - P(Ri | Rj) | / P(Ri))
= *dependency(Rj, Ri)*

Non-transitivity: Reasoning by contradiction. Let's assume that dependency relation is transitive. Let's take three rules Ri, Rj, Rk with *dependency(Ri, Rj)* = $\alpha/2 < \alpha$, *dependency(Rj, Rk)* = $\alpha/2 < \alpha$ and d*ependency(Ri, Rk)* = $c < \alpha$, then by transitivity *dependency (Ri, Rk)* = $\alpha/2 + \alpha/2 = \alpha$ and dependency (Ri, Rk) = $c < \alpha$. Then we have a contradiction with $\alpha < \alpha$. The dependency is not transitive.