A quick Documentation for Aris Implementation

We describe different packages and their important classes. The most important classes and functions are described briefly.

Package problemAnalyzer:
This is the main package responsible for parsing the problems and extracting quantities and entities. It is also the starting point for running Aris.

Class Entity:
This class is used to extract an entity. This can be done based on a *governor* or *dependent* in a relation or by just providing the index of the entity. Noun Modifers and Adjective modifiers and some other properties are also set for the entity.

Class QuantitativeEntity:
This class makes a quantitative entity based on one of the two options described for class Entity. It has an Entity inside and also a field named *num,* which is the number associated to the entity (can be null). It can be then modified to become an expression. This class sets some other important features for the quantitative entity including:

- *isQuestion*: Is the qentity a question entity?
- *superlative*: is it a superlative question?
- *fakeEntity*: If the entity does not correspond to a number, it is fake entity. For example: “He gave some of them to Sara”.
- *Path to verb*

Class AF:
This class extracts the information of one of the real sentence of the problem. It extracts the subject, indirect object, time, place, verb, relations of the verb according to the parser, a flag determining if the sentence has passive sound and a collection of quantitative entities (usually one). The name AF stands for Action (non-zero verbs in the paper) or Fact (Observation in the paper).

Class AFSentenceAnalyzer:
It receives a sentence as input, parses it and makes the AFs and quantitative entities and the question entity. It stores some properties for the sentence (e.g. the lemmas) that are used by other classes.

Class QuestionAnalyzer:
This class receives a whole problem and does the preprocessing on that (dealing with different formats of numbers, co-reference resolution for the pronouns). Then the AFs and entities for all sentences are extracted and stored as a whole. The sentences that do not have entities or numbers are handled here. Especial cases for “one” and “two” are also handled here in function postProcessQuestion. (Please refer to function for details). The code to determine if we should check features or not while matching the question with sentences is also written here as shouldIgnoreFeatures.

Package verbAnalyze:

In this package, the code for predicting sentence labels is written. Previously, we had verb prediction in VerbAnalyzer, but now we use SentenceAnalyzer. The main class is SentenceAnalyzer. In the function generateArffWithFeatures, an arff file is made for the instances based on the input feature types. Then, function analyzeArff is called to make classifiers and evaluate them.

Splitting the data into train and test is also done here. There is a boolean named carefulsplit. If we do not want identical verbs in both train and test sets, this variable should be true. There is also a variable to determine if the split should be done randomly or not. When this function is executed, two files are made with names pverbs_h1.txt and pverbs_h2.txt which will be then used in MathCoreNLP. If we want MathCoreNLP to use the predicted verbs, its PREDICT variable should be true, otherwise, gold standard verbs will be used.

There are three functions to make similarity-based, structural and lexical features. For the features that are verb based, a function named makeSentenceFeatures is called to make a copy of the features of each verb for every sentence with that verb. Something that you do not need to worry about is that 10 different arff files are made. This is because we had a feature selection method. We wanted the selected features just based on the training data. So, we did the feature selection 10 times. But you can simply forget it based on our experiments.

Running the parser was relatively time-consuming, and we required running it both for feature extraction (structure features) and solving the questions. For the first task, we ran the code just once to make a json file (See json generator). Then used it for structural feature extraction. So, if you changed the dataset, then you need to run MathJsonGenerator again to make that json file and then use it as input for the SentenceAnalyzer class. It also made the code a little confusing for splitting the data into train and test. Since it is required to know the number of quantitative entities of each dataset, but it can be computed while running the MathJsonGenerator code. The number of quantitative entities for each dataset is stored in a numCentsInDS variable which you should take care of to get exact results!

The best files to use is in folder DfN which I have sent to Nate. The folder prev contains our first version and the other files are the final version of the dataset (which was easier for us). Currently, other files, like a.txt, aa.txt, a_refined.txt, … are used in the code. The files with names including _refined refer to the final version of the datasets. I think it is best to get rid of unnecessary files at first. In general, DS1 corresponds to a.txt and aa.txt, DS2 corresponds to ixl_a.txt and ixl_aa.txt and DS3 corresponds to a2.txt and aa2.txt.

If you can run function test1() of this class, it means the code is running greatly!

Package equationExtraction:

Class ActionSpecifier:

This class is associated with each of the sentence fragments. It extracts holders and also applies circumscription assumption. The match function is also implemented here.

Class State:

This class makes a state given a quantitative entity. The expression of the state is also made here.

Class World:
This class makes all the states and then in function solveQuestion, solves the question based on different situations for question verbs. The function for finding the variable is also implemented in this class.

Python code:
There is a small python code to generate Resnik similarities. If you want to add new verbs, you may need that. Its name is verbPolarity2.py.

You will need some jar files to run the code. They can be found online. If you have difficulty finding these jar files or running the code, please send me an email. I think there are a lot of details that I did not mentioned.

Finally, there are some implementations done which we did not mention in the paper:
1) A quick solution for superlative problems (Difference of two numbers!)
2) Changing the sign of verb when we see “total of” in a sentence!
3) Buy and purchase verbs are labeled differently if there is $ in the question. Since when we buy something, we give money and receive an item.