RParse -- Extracting Structured Information from Materials Science Literature

Ziqin Rong, Dec 7th, 2015

Abstract

In Materials Sciences Research Community, one big painful process for research scientists is to search through a huge amount of literature to find a synthesis process for certain material system. It will be nice to have a database where researchers can just query certain material and find a visualized synthesis workflow directly. The goal of this project is to implement a machine learning system to digest Materials Science literature texts and extract structured information from them. For each sentence, we first classify whether that sentence tells a action like $input \xrightarrow[method]{verb} output$ or it only provides extra information for previous actions. Then, for the sentence which contains action information, substrings or words are extracted to fill 3 different fields:

1. key action verb
2. method argument associated with action verbs
3. input/output arguments

For field 1 and field 2, we use heuristic rules built from 3 papers and achieve a F1 score of 0.769 (verbs) and 0.724 (method arguments) on 14 test papers. For field 3, we used maximum entropy model and achieved an F1 score of 0.481.

Problem Statement

In Materials Sciences Research Community, one big painful process for research scientists is to search through a huge amount of literature to find a synthesis process for certain material system. It will be nice to have a database where researchers can just query certain material and find a visualized synthesis workflow directly.

On the other hand, the language describing these synthesis processes are generally instructional language, which in theory can be automatically digested by algorithms to generate action graphs. Leveraging the relative easiness for handling imperative language, similar tasks have been done to extract ingredients and instructions out of cooking recipes. [1]

However, it is not trivial to replicate what has been done in cooking recipes to Materials Science literature, mainly because the speciality of research language as well as the existence of special symbols (chemical formulas, units, etc.)

The goal of this project is to implement a machine learning system to digest instructional texts and extract structured information from them. These texts are paragraphs from literatures describing the synthesis process of certain material system. An example of such parsing can be seen in the following section.

Dataset

The training corpus comes from the papers on ceder group publications and some papers provided by Prof. Elsa Olivetti. An example input paragraph will be like:

A typical synthesis of Na3MnPO4CO3 is as follows: Separately, 0.02 mol of Mn(NO3)2·4H2O was dissolved in 50 mL of water to form a clear solution (A). A total of 0.02 mol of (NH4)2HPO4 and 20 g of Na2CO3 were dissolved in 100 mL of water to form a clear solution (B). Solution A was then quickly added to solution B under fast magnetic stirring. The obtained slurry was then transferred to a glass bottle sealed with a cap. The bottle was heated in a 120 °C oil bath in an Ar flushed glovebox for 4–72 h, after which it was taken out of the oil bath and

slowly cooled down to room temperature. The slurry was centrifuged and washed with distilled water and methanol several times to separate the solids. The solid samples were dried in a vacuum oven at 40 °C overnight.

Parsing this paragraph manually, we are able to extract structured information from each sentence:

1. **sen1**:
    - **action1**:
        - **v**: 'dissolved'
        - **input1**: '0.02 mol of Mn(NO3)2·4H2O'
        - **output1**: 'a clear solution (A)'
        - **method1**: 'in 50 mL of water'
2. **sen2**:
    - **action1**:
        - **v**: 'dissolved'
        - **input1**: '0.02 mol of (NH4)2HPO4'
        - **input2**: '20 g of Na2CO3'
        - **output1**: 'a clear solution (B)'
        - **method1**: 'in 100 mL of water'
3. **sen3**:
    - **action1**:
        - **v**: 'added'
        - **input1**: 'Solution A'
        - **output1**: 'implicit object'
        - **method1**: 'under fast magnetic stirring'
        - **method2**: 'quickly'
        - **method3**: 'to solution B'
4. **sen4**:
    - **action1**:
        - **v**: 'transferred'
        - **input1**: 'The obtained slurry'
        - **output1**: 'implicit object'
        - **method1**: 'to a glass bottle sealed with a cap'
5. **sen5**:
    - **action1**:
        - **v**: 'heated'
        - **input1**: 'the bottle'
        - **output1**: 'implicit object'
        - **method1**: 'in a 120 °C oil bath in an Ar flushed glovebox'
        - **method2**: 'for 4–72 h, after which it was taken out of the oil bath and slowly cooled down to room temperature.'
6. **sen6**:
    - **action1**:
        - **v**: 'centrifuged, washed'
        - **input1**: 'the slurry'
        - **output1**: 'implicit object'
        - **method1**: 'with distilled water and methanol several times'
7. **sen7**:
    - **action1**:
        - **v**: 'dried'

        - **input1**: 'The solid samples'
        - **output1**: 'implicit object'
        - **method1**: 'in a vacuum oven'
        - **method2**: 'at 40 °C overnight'

If there are multiple actions in one sentence, such as the sentence: ***The mixture was then manually mixed with polytetrafluoroethylene (PTFE) and rolled into thin film.*** Then the annotation will be something like

- **sen4**:
    - **action1**:
        - **v**: 'mixed'
        - **input1**: 'The mixture'
        - **output1**: 'imiplicit object'
        - **method1**: 'manually'
        - **method2**: 'with polytetrafluoroethylene (PTFE)'
    - **action2**:
        - **v**: 'rolled'
        - **input1**: 'imiplicit object'
        - **output1**: 'imiplicit object'
        - **method1**: 'into thin film'

Two more things worth mentioning:

- For some sentences, they only provide extra information rather than a action step, then such sentence will be tagged as *extra information*, the annotation will be:
- **sen5**:
    - **state**: extra information
- For each sentence which is not tagged as *extra information*, there must exist at least one input and one output argument for every action, if there is no such corresponding substrings in the original sentences, input and output arguments will be annotated as *implicit object*

Similar to the above examples, we manually annotated 17 papers as our working dataset. All the data can be found at data repo. For every paper, the original text is in the file ****.raw.txt***, its manual gold annotation is stored in corresponding ****.gold.yaml***, the RParse algorithm parsed data is stored in ****.RParse.yaml***

Method and Results

Digest on a sentence basis

As demonstrated in the previous section, we digest the text on a sentence basis, for each sentence, we aim to extract 4 different domain information, i.e. *key action verb*, *input argument*, *output argument*, *method argument*, such each sentence express a transition like

$$input \xrightarrow[method]{verb} output$$

Extracting Action Verbs and Method Arguments using heuristic rules

We use heuristic rules to extract action verbs and associated method arguments. These heuristic rules were established when creating gold annotation for the first 3 papers. They are then evaluated over the extra 14 papers to ensure their effectiveness. The results are shown below.

These heuristic rules are based on sentence tree parsing and dependency parsing. We first generally state the rules

below and then show several examples to demonstrate them.

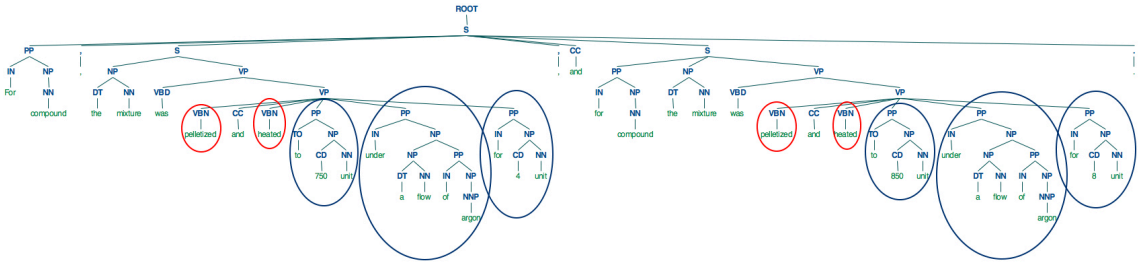
Rules for extracting action verbs

- Find the word with a part of speech tag **VBN** at the highest level of parse tree. This verb is the key action verbs.
- If there are multiple verbs at the same level, then we check if these verbs have the same parent node. If they belong to the same parent node, they belong to the same action group, if they have different parent nodes, they belong to different action groups (example in previous section, action1 and action2). The verbs that appear early in the sentence has a smaller action group sequence number.

Rules for extracting method arguments

- method arguments are the **PP**, **RB** or **ADVP** phrases that belong the same parent node of the key action verb
- if there is VBG which has a *xcomp* dependency relation to the action verb, the VP that VBG is in is also acting as a method argument

Demonstration examples 1

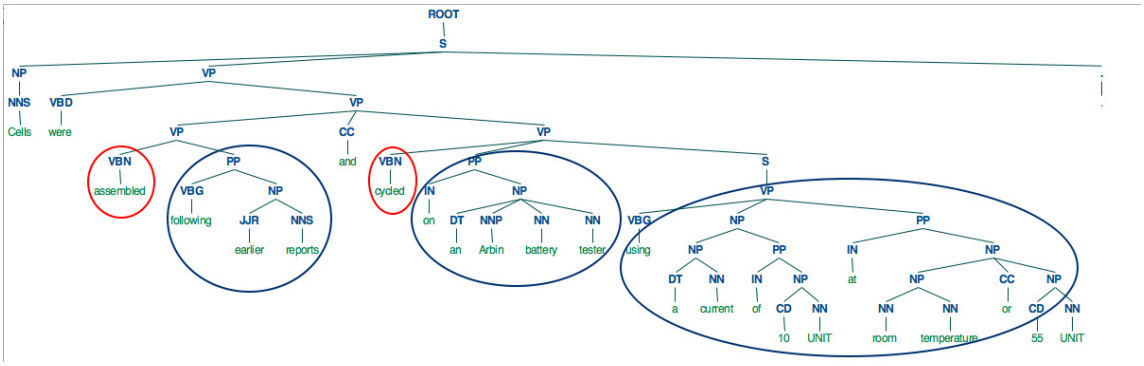


Original Text: For Na3V2(PO4)2F3, the mixture was **pelletized and heated** *to 750 °C under a flow of argon for 4 h*, and for Na3GaV(PO4)2F3 the mixture was **pelletized and heated** *to 850 °C under a flow of argon for 8 h*.

For the example above, according to the verb extraction rules, there are 4 VBNs at the same level. But there are two parent nodes for these verbs, so there are two action verb groups, each of which have two verbs, and for each action group, there are three method arguments (3 PPs) associated with it, because these PPs share the same parent node with the action verbs. thus, *RParse.py* will parse the sentence into the following format:

- **action1**:
  - **v**: 'pelletized, heated'
  - **input1**: 'the mixture'
  - **output1**: 'Na3V2(PO4)2F3'
  - **method1**: 'to 750 °C'
  - **method2**: 'under a flow of argon'
  - **method3**: 'for 4 h'
- **action2**:
  - **v**: 'pelletized, heated'
  - **input1**: 'the mixture'
  - **output1**: 'Na3GaV(PO4)2F3'
  - **method1**: 'to 850 °C'
  - **method2**: 'under a flow of argon'
  - **method3**: 'for 8 h'

Demonstration examples 2



Original Text: Cells were **assembled** *following earlier reports* and **cycled** *on an Arbin battery tester using a current of 10 mA/g at room temperature or 55 °C*.

For the example above, there are two action groups according to the verb extraction rule. Besides all the PP acting as method arguments, the VP phrase *using a current of 10 mA/g at room temperature or 55 °C* is also a method argument following the 2nd rule in identifying method arguments, because of dependency relationship `((u'cycled', u'VBN'), u'xcomp', (u'using', u'VBG'))`.

Results (Key Action Verbs & Method Arguments)

Implementing the above heuristic rules, we evaluate the effectiveness of parsing on key action verbs and method arguments. Detailed steps below:

1. Enter the root directory of RParse Repo (make sure you installed the dependencies and run `python setup.py develop`, details about dependencies can be found in **Appendix: Code Implementation**)
2. `cd misc`
3. `python RPrase_predict_folder.py -f ../data/verb_method_arg/train` to read all ***.raw.txt** files and generate corresponding ***.RParse.yaml** files. Under folder `../data/verb_method_arg/train` are 3 papers which we used to establish the heuristic rules
4. `cd ../misc/evaluation_scripts`
5. `python evaluate_action_verbs.py -f ../data/verb_method_arg/train` to compare ***.RParse.yaml** files with gold annotation ***.gold.yaml** files, evaluate the *recall, precision and F1 scores* of key action verbs
6. `python evaluate_method_arg.py -f ../data/verb_method_arg/train` evaluate the *recall, precision and F1 scores* of method arguments.

The results are:

- **recall** for key action verbs is **0.870**
- **precision** for key action verbs is **0.952**
- **F1 score** for key action verbs is **0.909**
- **recall** for method arguments is **0.647**
- **precision** for method argument is **0.667**
- **F1 score** for key method argument is **0.657**

Similar procedures can be done on the test papers dataset (14 papers used to test heuristic rules), from the root directory of the repo

1. `cd misc`

2. `python RPrase_predict_folder.py -f ../data/verb_method_arg/test`
3. `cd ../misc/evaluation_scripts`
4. `python evaluate_action_verbs.py -f ../data/verb_method_arg/test`
5. `python evaluate_method_arg.py -f ../data/verb_method_arg/test`

The results are:

- **recall** for key action verbs is **0.783**
- **precision** for key action verbs is **0.755**
- **F1 score**for key action verbs is **0.769**
- **recall** for method arguments is **0.669**
- **precision** for method argument is **0.770**
- **F1 score** for key method argument is **0.716**

These results on test dataset show that the heuristic rules generalize quite well on identifying key action verbs and associated method arguments.

Extracting input output arguments using Maximum Entropy Model

Here we detail our method in finding input and output arguments. We aim to build a machine learning pipeline. The first classifier read original text from paper and extract input and output arguments together, the second classifier classify these extracted phrases into input and output arguments respectively.

This pipeline system makes sense as the input arguments of one sentence can be output of the other sentence, thus linguistically input and output arguments are symmetric. It is much easier to extract them together than extract input output separately.

Due to the time constrain of this project, we only implement and report the first classifier here.

Heuristic Rules Baseline Test

Extracting input output arguments is not as easy as action verbs and method arguments. We here run a baseline test using heuristic rules extracting input output arguments.

As all of the input and output arguments in gold annotations are Noun Phrases in the parsing tree, thus, for each sentence, we extract all of the NPs from sentence parse tree, and identify them as the input and output arguments. Compare this baseline algorithm with the gold annotations:

1. `cd misc`
2. `python ioput_baseline_predict_folder.py -f ../data/input_output_arg/test`
   ioput_baseline_predict_folder.py is the script implementing this heuristic baseline rule across all *.raw.txt* files in a folder.
3. `cd ../misc/evaluation_scripts`
4. `python evaluate_ioput_baseline.py -f ../data/input_output_arg/test/`

The results are:

- **recall** for input output argument is **1.0**
- **precision** for input output argument is **0.0342**
- **F1 score** for input output argument is **0.0661**

The baseline test shows that heuristic rules can't achieve a satisfactory performance. The recall score of baseline test is really high, this is because all of the input output arguments in gold annotations are NPs (as expected).

Unigram Maximum Entropy Model for classifying NP phrases

Thus, we construct a maximum entropy model to classify whether a NP phrase is input output argument or not.

This is supervised machine learning algorithm and we need to construct a training dataset.

Step 1: First step is to split all papers randomly to treat part of them as training data and part of them as testing data. This is done by script `misc/data_split_input_output_arg.py`. The script splits the data and move 12 papers to folder `data/input_output_arg/train` and 5 papers to folder `data/input_output_arg/test`.

Step 2, we construct the training dataset by running script `construct_ioput_train.py`. What this script does is to read all of the NPs of sentence in the parse tree, and compare against the gold annotation, if this NP is an input output argument according to the gold annotation, then we tag it as **input_output** class, if not, we tag it as **else** class. The results of this script is saved in file `ioput_train_db.yaml`

Step 3: script `train_ioput_maxent_unigram.py` constructs the features of each NP using the unigram maximum entropy model and train a MaxEnt classifier, the classifier is saved in `ioput_maxent_classifier.pickle` file. How to construct the unigram feature from a phrase can be referred at *RParse._maxent_unigram_feats* function in *RPasre.py* module.

Step 4: *RParse.parse_input_output* module will read `ioput_maxent_classifier.pickle` file and use to classify the new NPs as *else* or *input_output* tag.

We evaluate the performance of extracting input output arguments:

1. `cd misc`
2. `RPrase_predict_folder.py -r ../data/input_output_arg/test`
3. `cd ../evaluation_scripts/`
4. `python evaluate_ioput_arg.py -f ../data/input_output_arg/test/`

Results:

- **recall** for input output argument is **0.5**
- **precision** for input output argument is **0.464**
- **F1 score** for input output argument is **0.481**

The result shows a huge improvement on the performance by constructing a maximum entropy model.

Similar procedures is done on the training dataset, results:

- **recall** for input output argument is **0.568**
- **precision** for input output argument is **0.588**
- **F1 score** for input output argument is **0.5777**

Note: There are some subtleties in the code implementation. The classifiers are trained on the phrases before *Postprocess*, so common words from *PreProcess* such as **COMPOUND UNITS** can be leveraged.

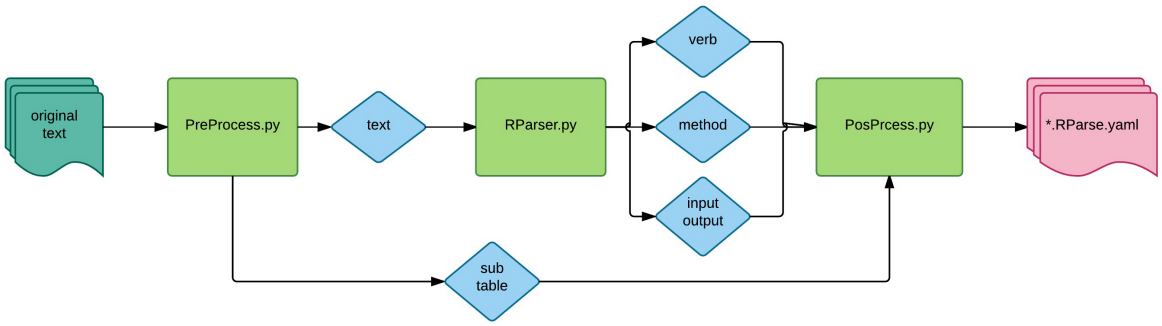
Better MaxEnt Model

Due to the time constraint and resource limitation, I can only personally annotate 14 papers, I assume with more data, this Maximum Entropy model can work much better. It is also possible that we construct bi-gram or trigram features from NP with more data.

Appendix: Code Implementation

Run `python setup.py develop` after downloading the repo from MIT repo or Personal repo

Architecture



The code architecture is shown in the figure above. The original text is firstly digested by a *PreProcess.py* module. The existence of Preprocess module is to substitute some special words in the text which are hard to work on for the *RParse.py* module. For example, all chemical formulas such as **Na3MnPO4CO3** is substituted by word **COMPOUND**. The *PreProcess.py* module will then the processed text and create a substitute table, in which all such **Na3MnPO4CO3** $\rightarrow$ **COMPOUND** substitution is recorded. The processed text is passed to *RParse.py* model, which extracts domain specific substrings, these substrings still contain substituted words such as **COMPOUND** etc. They are passed together with the *sub table* to *PostProcess.py* which will recover the original words from substituted ones. These final results are written in the ***.RParse.yaml*** files.

Codes

PreProcess.py

Several main problems solved by *PreProcess.py* module is

- **encoding problems**: some symbols are not recognizable by normal *utf-8* encoding, we resolve this issue in PreProcess module by unicode coding
- **chemical formula**: chemical formulas are hard to run with parsing algorithms, as one single chemical formula will be parsed into multiple word tokens, such as the chemical formula blow. PreProcessor will turn these words into a single word **COMPOUND**.

$$Na3V2(PO4)2F3 \rightarrow [Na3V2, (, PO4,), 2F3]$$

- **Units**: some units are hard to run with parsing algorithms, one single unit word gets parsed into multiple tokens, such as the example below. PreProcessor will turn these words into a single word **UNIT**.

$$mAh/g \rightarrow [mAh, /, g]$$

- **Number Range**: some number ranges are hard to run with parsing algorithms, one single unit word gets parsed into multiple tokens, such as the example below. PreProcessor will turn these words into a single word **NUMBER**.

$$4 - 72 \rightarrow [4, -, 72]$$

- etc. There are multiple other things *PreProcessor.py* takes care of. *PreProcessor.py* mainly uses heuristic rules for detecting these special words and substitute them correspondingly.

RParse.py

RParse has two key functions:

- *parse_v_method* returns key action verb groups and associated method arguments using heuristic rules
- *parse_input_output* returns noun phrases tagged as **input_output**

PostProcess.py

- recover the original substrings using substitute table from the output of *PreProcess.py*.
- recover some special symbol and comma, periods position.
    - To parse the original sentence correctly, *PreProcessor.py* will always put an extra space between word and comma or period, *PostProcessor.py* recovers this change
    - In parsing, **'('** will be notified as special **'-LRB-'**, and **')'** will be notified as special **'-RRB-'**, *PostProcessor.py* recovers this change
- etc.

Evaluation Scripts

Evaluation scripts are written to compare all of the ***.RParse.yaml*** files with corresponding ***.gold.yaml*** annotations to evaluate recall, precision, F1 scores of each specific domains.

evaluate_action_verbs.py

- Takes a file folder as input and compare all matched ***.RParse.yaml*** and ***.gold.yaml*** files under that file folder to evaluate the effectiveness of extracting key actions verbs.
- example: `python evaluate_action_verbs.py -f ../data/verb_method_arg/test`

evaluate_method_arg.py

- Similar to *evaluate_action_verbs.py*, but evaluate the effectiveness of extracting associated method arguments.
- example: `python evaluate_method_arg.py -f ../data/verb_method_arg/train`
- etc.

Dependencies

This code repo replies heavily on the standford NLP codes, which is interfaced within NLTK package. The way to configure standford codes

1. download it from link
2. Create a new folder (*stanford_parser_folder* in my environ.yaml file). Place the extracted files into this jar folder: stanford-parser-3.x.x-models.jar and stanford-parser.jar.
3. Open the stanford-parser-3.x.x-models.jar using command `jar xf jar-file [archived-file(s)]`
4. Browse inside the jar file; edu/stanford/nlp/models/lexparser. Again, extract the file called 'englishPCFG.ser.gz'.
5. Copy 'englishPCFG.ser.gz' into a different folder and copy its path to *model_path* in environ.yaml file.

---

1. New York Times Cooking Recipe Database is one example, Los Angeles Times Cooking Recipe Database is another↩