



TMOP: Translation Memory Open-source Purifier

Version: 1.2

Authors:

Masoud Jalili Sabet

Matteo Negri

Marco Turchi

November 28, 2016

Contents

1	Introduction to TMOP	1
2	User's Guide	2
2.1	Requirements	2
2.2	Usage	3
2.2.1	First Run	3
2.2.2	Configuration File	4
2.2.3	Input File	6
2.2.4	Output Files	7
2.3	Adding a new filter	8
2.4	Adding a new decision policy	9
3	Developer's Guide	10
3.1	How filters work	10
3.2	TU class	11
3.3	Writing a new filter	12
3.3.1	Initialize	13
3.3.2	Process TU	14
3.3.3	Finalize a full scan	15
3.3.4	Finalize	15
3.3.5	Decide	16
3.4	Writing a new policy manager	16
3.4.1	Initialize	17
3.4.2	Decide	17
4	Implemented Filters	19
4.1	SampleFilter	19
4.2	LengthRatio & ReverseLengthRatio	19
4.3	WordRatio & ReverseWordRatio	20
4.4	RepeatedChars	20
4.5	RepeatedWords	21
4.6	WordLength	21
4.7	TagFinder	22

4.8	Lang_Identifier	22
4.9	AlignedProportion	22
4.10	BigramAlignedProportion	23
4.11	NumberOfUnalignedSequences	23
4.12	LongestAlignedSequence & LongestUnalignedSequence	24
4.13	AlignedSequenceLength & UnalignedSequenceLength	24
4.14	FirstUnalignedWord & LastUnalignedWord	25
4.15	WE_Average & WE_Median	25
4.16	WE_BestAlignScore	26
5	Implemented Policy Managers	27
5.1	OneNo policy manager	27
5.2	MajorityVoting policy manager	27

List of Code Snippets

2.1	The config file in JSON format	4
2.2	The input file sample	6
2.3	The input & output entries in the config file	7
2.4	The filters in the config file	7
2.5	Add filter to the config file	8
3.1	The TU class	12
3.2	Example, TU alignment attribute	12
3.3	Abstract filter class	12
3.4	Example, initialize method	14
3.5	Example, process_tu method	15
3.6	Example, finalize method	15
3.7	Example, decide method	16
3.8	Abstract policy manager class	16
3.9	Example, OneNo policy manager	17

Chapter 1

Introduction to TMOP

Translation memories (TMs) are collections of (source, target) segments, called “translation units” (TUs), which are typically used to aid human translators operating in a computer-assisted translation (CAT) framework. Their use is based on computing a “fuzzy match score” between an input segment to be translated and the left-hand side (the source) of each TU stored in the TM. If the score is above a certain threshold, the right-hand side (the target) is presented to the user as a translation suggestion. In such framework, when translating a document, the user can store each translated (source, target) pair in the TM for future use. Each newly added TU hence contributes to the growth of the TM which, as time goes by, should in principle become more and more useful to the user.

The public TMs are fed with TUs coming from multiple users and typically grow in a less homogeneous way. Together with the quantity, the quality of the stored material is a crucial factor that determines the usefulness of TM itself to support the translation process. Indeed, the continuous maintenance of the TM should go hand in hand with its continuous growth. Focusing on the maintenance aspect, we developed an open-source software called “Translation Memory Open-source Purifier” (TMOP).

TMOP is an open-source software written in Python and it is useful for cleaning and maintaining a TM. The TMOP goal is removing bad TUs and poor translations from the TM and make it more suitable for translation tasks.

Chapter 2

User's Guide

TMOP consists of three parts, the core, filters and policy managers. **Core** is the main part of the code and it manages the workflow between filters, policy managers and the files. **Filters** are responsible for detecting bad TUs. Each filter can detect a specific type of problems (*e.g.* formatting, fluency, adequacy) in a TU. The **policy managers** take the individual results from each filter and decide whether keeping or removing each TUs based on different possible strategies. This chapter discusses about how to use the TMOP and how to change the configuration for the mentioned parts of the program.

2.1 Requirements

The TMOP is written in Python(2.7.10), so the first thing needed is **installing Python**.¹ Some filters may use other python libraries. It is commended to use **pip**² for installing these libraries. pip will install automatically if the latest version of Python is installed on the computer.

For Microsoft Windows: The Windows users should check if Python works in the command prompt (CMD). It should recognise “python” as a command and open a Python command line. If Python is not working in CMD, it should be added to Windows path. Here are the instructions:

1. Go to Computer Properties (System in Control Panel).
2. Go to Advanced System Settings.
3. Click Environment Variables.
4. Append your Python folder (ex: “;C:\python27”) to the Path variable.

¹<https://www.python.org/downloads/>

²<https://pip.pypa.io/en/latest/installing.html>

5. Restart CMD.

The TMOP is tested on different operating systems and it properly works in Linux, Windows and Mac OS.

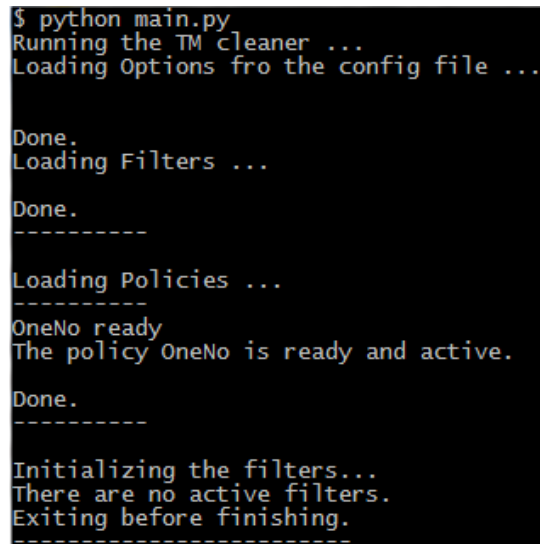
2.2 Usage

2.2.1 First Run

Open a terminal or CMD and go to the main folder of the project. There is a file named *main.py* in the project folder. Try to run the code with this command:

```
> python main.py
```

The output of the program should be like figure 2.1.



```
$ python main.py
Running the TM cleaner ...
Loading Options fro the config file ...

Done.
Loading Filters ...

Done.
-----
Loading Policies ...
-----
OneNo ready
The policy OneNo is ready and active.

Done.
-----
Initializing the filters...
There are no active filters.
Exiting before finishing.
-----
```

Figure 2.1: First Run

In the output of the program in the terminal there are three loading progresses about **Config File**, **Filters** and **Policies**. The configuration file contains the settings of the program such as input files, output files and filters. The TMOP uses filters to detect bad TUs. Each filter detects a certain kind of problem and can *accept* or *reject* the TU. After all the filters return their answers for each TU, the decision policies will decide to keep or remove the TU based on filters outputs. There is a thorough documentation about these parts in chapter 3.

2.2.2 Configuration File

The config file is in **JSON**³ format and it looks like the code snippet 2.1.

Code Snippet 2.1: The config file in JSON format

```
1 {
2   "options": {
3     "input file":          "sample_en_it.csv",
4     "align file":          "sample_align",
5     "token file":          "sample_token",
6
7     "output folder":       "output",
8
9     "source language":     "en",
10    "target language":      "it",
11
12    "normalize scores":     "true",
13    "emit scores":          "false",
14    "no out files":         "false",
15    "max decision":         -1
16  },
17
18  "policies": [
19    ["OneNo",               "on"],
20    ["TwentyNo",           "off"],
21    ["MajorityVoting",      "off"]
22  ],
23
24  "filters": [
25    ["SampleFilter",        "off"],
26    ["LengthRatio",         "off"],
27    ["ReverseLengthRatio",  "off"]
28  ]
29 }
```

This file has three parts named *options*, *policies* and *filters*. By changing these settings and running the *main.py* code, TMOP will produce different outputs and information.

In the root folder of the project, there is a folder named **data**. The input data (the input TM that has to be cleaned) should be copied to this folder. After that, the name of the TM file should be written in the **input file** field in the *config file*. The *config loader* in TMOP will search the *data* folder for

³<https://en.wikipedia.org/wiki/JSON>

a file with the name in *input file* field. Usually these input files have a **file extension** in their names (like ".csv", ".txt", ".in", etc.). **It is important** to put the file extension in the file name (like "input_tm.csv") for the "input file" field in the config file. There are also two fields for the **source language** and **target language**. These fields show the expected languages of the source and target segments and they are in two-letter codes (eg. "en", "it", "de", "fr").

The **token file** field is for providing tokenized version of the source and target sentences to the cleaner. Some filters may use the tokenized version of the source and target sentences. If the **token file** is not available, the TMOP will use a simple regular expression and do the tokenization (with a performance overhead). So, this field is not necessary and this line can be removed if not available. The token file format is similar to the input file. It has one TU per line and each TU is made of a source and a target sentence, separated by a tab character. The TUs in the token file and the input file should be correspondent.

The **align file** field is for providing alignment info to the cleaner. It is not necessary and this line can be removed, if there are not filters that use the alignment info. In the alignment file, each line has the alignment information for the corresponding TU in the TM. In each line, there is a list of ID pairs (i-j), where a pair i-j indicates that the *i*th word of the source sentence is aligned to the *j*th word of the target sentence. This format is known as "Pharaoh format" and it can be produced by two widely used word aligners: mgiza++⁴ and fastalign⁵. In the *data* folder, there is an example file named "sample_align" that shows the word alignments of the sample input TUs.

For improving the performance of the TMOP and for research purposes, there are other options in the config file for controlling the production of the output files. For this reason the following fields have been added: **no out files**, **max decision** and **emit scores**.

If the **no out files** field is set to true, the TMOP will not generate the accept/reject files (Only log file will be generated). The default value for this field (even without mentioning in the config file) is "false".

The **max decision** field indicates the number of TUs (starting from the first one) to process in the decision section. With this option it is possible to clean a small part of the TM and check the results (for tuning the filters and policies). If this field is set to -1 (default value), the entire TM will be processed.

If the **emit scores** field is set to true, the TMOP will generate a new file in the *output folder* containing scores from all filters for each TU. In this file, each line represents a TU and contains a comma-separated list

⁴<https://github.com/moses-smt/mgiza>

⁵https://github.com/clab/fast_align

of scores. These scores are calculated by the active filters in the *learning* section. The default value for this field (even without mentioning in the config file) is "false".

2.2.3 Input File

The Input file is a text file that has one TU per line. Each line is made of three parts: the TU ID, the source sentence and the target sentence. These three part are separated by tabs.⁶ An example of a line in the input file is shown in code snippet 2.2.

Code Snippet 2.2: The input file sample

834	Source sentence	Target sentence
-----	-----------------	-----------------

To allow TMOP to read the input file, first, the file should be put in the "TMOP/data/" folder like in figure 2.2. It is important to note that **the encoding of the file should be "UTF-8"**. A toy input file named "sample_en_it.csv" is available in the "TMOP/data/" folder.

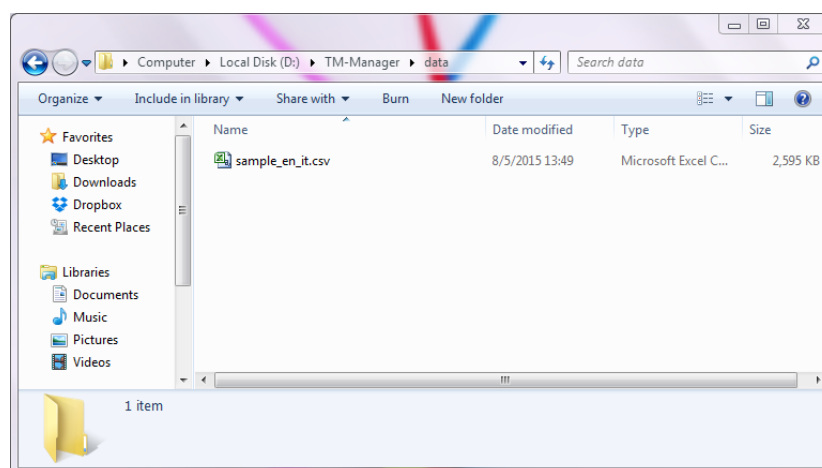


Figure 2.2: data folder for input files

After that the "TMOP/config.json" file should change like this code snippet 2.3.

⁶Tabs are not allowed in the source and target sentences

Code Snippet 2.3: The input & output entries in the config file

```
2      "options": {  
3          "input file":          "sample_en_it.csv",  
4          "output folder":      "output",
```

In this step the program should work and by turning on some filters like code snippet 2.4, it should create some outputs in the *output folder*.

Code Snippet 2.4: The filters in the config file

```
14      "filters": [  
15          ["SampleFilter",      "on"],  
16          ["LengthRatio",      "off"],
```

2.2.4 Output Files

After running the code, TMOP will produce some files in the **output folder**. These files have similar names to the input TM file. So by running the code with different inputs, the outputs will not overwrite each other.

AS an example, by running the TMOP with the “*sample_en_it.csv*” file and turning on the “*SampleFilter*”, it will produce four files in the **output** folder. The TMOP first checks the TUs and tries to parse them. If the TU has a problem (like different encoding), TMOP will skip the TU. There is a file named “*skipped_sample_en_it.csv*” and it contains all the TUs that were skipped in the parsing phase.

Based on the *decision policy*, the TMOP will produce different number of output files. The most important ones are **accept** and **reject** files. In our example, the accept file is “*accept_OneNo_sample_en_it.csv*” and the reject file is “*reject_OneNo_sample_en_it.csv*”. the “OneNo” policy will check all the filters answers for a TU and it will put the TU in the reject file, if one or more filters reject that. Otherwise, it will put the TU in the accept file.

The last file in the *output* folder is the **log file** and in this example, it’s “*decision_log_sample_en_it.csv*”. It contains all the decisions that were made during the cleaning process. For each TU, there is a line in the *log file* that shows the results of active policy managers for that TU. In the line, for each policy manager, a number and a string, separated by tab will be written in the log file. The number can be 0, 1 or 2, and the string is the decision of the policy manager. If the decision is “reject”, the number is 0. If the decision is “accept”, the number is 2 and for all the other decisions, the number is 1. The evaluators can use the numbers for calculating the performance of the cleaner, and the string is just for human readability. There is a sample log file in the output folder named “(expected) deci-

sion_log_sample_en_it.csv". The output of the first run should be similar to this file.

2.3 Adding a new filter

The TMOP is made in a way that adding and managing filters is really easy and fast. Adding a new filter simply can be done by copying it to the **filters** folder and changing the *config file*. This section describes an example of adding a new filter.

There are already some implemented filters in the filters folder and their descriptions are available in chapter 4. The other way of getting new filters is to download the filters that other developers on the Internet implemented. In this example, the sample filter is named "NewFilter".

Each filter **must have** a folder with the name of the filter and there should be a Python file inside the folder with the same name like shown in figure 2.3.

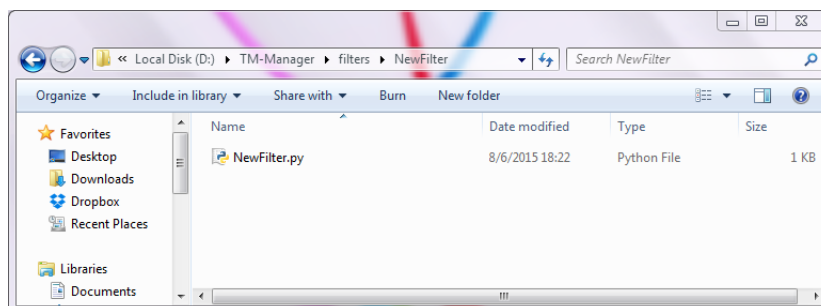


Figure 2.3: The folder of NewFilter

After downloading the new filter, put it in the "TMOP/filters/". The only thing left to do is adding the filter to the *config file*. As we discussed in subsection 2.2.2, there is a part in the config file for the filters. Another pair should be added to the filters like code snippet 2.5.

Code Snippet 2.5: Add filter to the config file

```
14     "filters": [  
15         ["SampleFilter",          "off"],  
16         ["ReverseLengthRatio",   "off"],  
17         ["NewFilter",            "on"]  
18     ]
```

Notice that at the end of last bracket there is no comma. After doing the things described above, the new filter should work fine. If it does not work,

look at the output error in terminal and check the config file. The config file should be in **JSON** format. It can be checked with an online [JSON parser](http://json.parser.online.fr/).⁷

2.4 Adding a new decision policy

Adding new policy is the same as adding new filters which we discussed in section 2.3. First the new policy should be copied to “*TMOP/policies/*”. The next thing is adding a new line to **policies** part of the *config file* with the name of the new policy (same as the filters). There are some policy managers in the “policies” folder that can be used for cleaning. The **default** policy manager is “OneNo”. It rejects a TU if there is even one filter that rejected the TU. The descriptions of policy managers are available in chapter 5.

⁷<http://json.parser.online.fr/>

Chapter 3

Developer's Guide

This chapter discusses the structure of TMOP, how filters are working and how to write new filters and policy managers. As mentioned in the introduction, the TMOP is written in Python (2.7.10) and its job is reading a TM and removing bad TUs. The filters are responsible for detecting bad TUs and in the next section the structure of filters is described.

3.1 How filters work

In TMOP, filters are responsible for detecting bad TUs. Filters can be analyzed as standalone programs and in that case they are also responsible for removing bad TUs and they have to organise the I/O. In this case, running more than one filter can be time-consuming and it is not efficient. So in this structure, filters are only responsible for detecting bad TU.

The filters can be written as standalone programs. In this way the program reads the TU, detect the problems and write the clean TM to the file. It can do the job in **two phases**. In the first phase the program can *learn* about the TM and gather some statistics (like calculating average of something in TUs). This phase can have more than one pass through the TM. In the second phase the program can use the gathered information and *decide* if a TU is good or not. A flowchart of this program is shown in figure 3.1.

As shown in figure 3.1, first the program should **initialize** its variables and after that start the learning phase. In the learning phase it has zero, one or more **iterations** over the TM. In each iteration the program passes through the TM and **process each TU**. After **finalizing** the learning phase the **decision** phase will start and it has one more pass through TM.

Instead of writing a standalone program for each filter, TMOP can be used. The TMOP job is managing the memory and I/O, and run all the filters independently at the same time. For each filter, developers should only

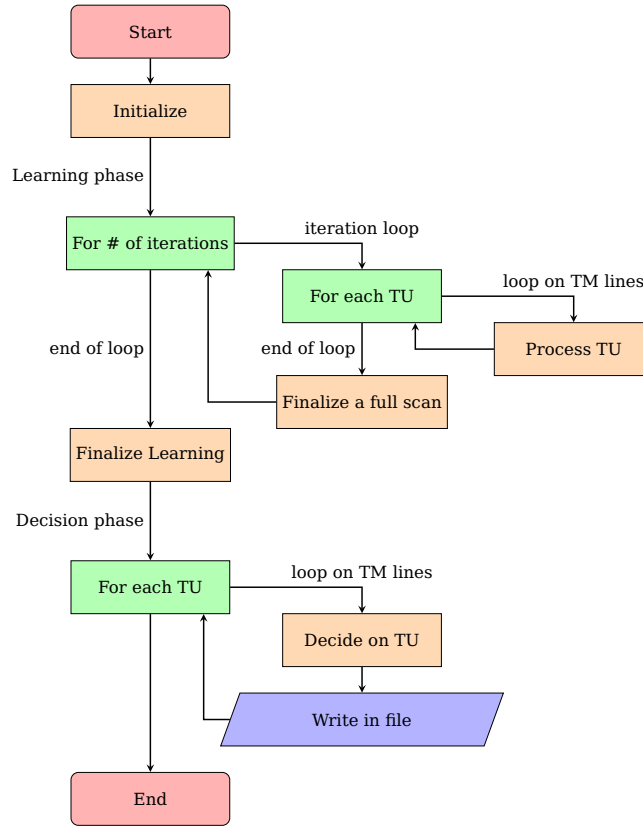


Figure 3.1: Simple program for a filter

write the orange boxes in the flowchart and the rest is up to the TMOP. In the filter class there are corresponding methods for these orange boxes and based on type of the filter, some of these methods can be left empty and do nothing.

3.2 TU class

The TMOP has to pass TUs to all active filters and give the results of filters to the policy managers. It should also decode the TUs and write them to files. Some filters need the **alignment information** or the **word tokens** of the TU. For making it easier and more readable, a class for representing a TU is implemented. It is a container of source and target phrases, the word tokens of source and target and the alignment of the source and target. The code is like code snippet 3.1.

Code Snippet 3.1: The TU class

```

1 class TU(object):
2     def __init__(self):
3         self.src_phrase = ""
4         self.trg_phrase = ""
5
6         self.src_tokens = []
7         self.trg_tokens = []
8
9         self.alignment = []

```

The “src_tokens” and “trg_tokens” attributes are lists of tokenized words. The alignment attribute is a list of aligned pairs. Each aligned pair is a pair of two numbers showing the index of aligned words in source and target phrases. The index is an integer number starting from zero. An example of the alignment attribute is shown in code snippet 3.2.

Code Snippet 3.2: Example, TU alignment attribute

```
self.alignment = [[0, 0], [1, 1], [2, 1], [3, 3]]
```

3.3 Writing a new filter

For better understanding of the code, we assume that the new filter is called “NewFilter” and it has to calculate the ratio of the length of the source phrase to the target phrase. If the ratio for a TU is different from the average, it will reject the TU. In each subsection below, after describing the methods, some part of this example filter will be written.

Each filter must have a folder with a Python file inside, **both with the name of the filter**. For using the new filter, it should be put in the filters folder of TMOP. For example, if the new filter is named “**NewFilter**”, there should be a folder like “*TMOP/filters/NewFilter/*” and a file named “*NewFilter.py*” should be inside of this folder. There can be other folders and files in this folder, but the Python file with the same name as the filter is necessary.

There is an abstract base class for filters named “*abstract_filter.py*” and it is located in “*TMOP/filters/*”. All the filters **should derive this class** and override its methods. As discussed in section 3.1, filters should have methods similar to orange boxes in figure 3.1. The **abstract_filter** class has these five methods. The code for the **abstract_filter** (without comments) is shown in code snippet 3.3.

Code Snippet 3.3: Abstract filter class

```

1 class AbstractFilter:
2     __metaclass__ = ABCMeta

```



```

3
4     num_of_scans = 0
5
6     def __init__(self):
7         pass
8
9     @abstractmethod
10    def initialize(self, source_language, target_language):
11        pass
12
13    @abstractmethod
14    def finalize(self):
15        pass
16
17    @abstractmethod
18    def process_tu(self, tu, num_of_finished_scans):
19        pass
20
21    @abstractmethod
22    def do_after_a_full_scan(self, num_of_finished_scans):
23        pass
24
25    @abstractmethod
26    def decide(self, tu):
27        return 'accept'

```

There is a variable named **num_of_scans** and five methods in the abstract class. The TMOP will call these methods for all active filters like the flowchart in figure 3.1. The **num_of_scans** variable indicates the number of scans needed for the filter and its **default value is “0”**. If the filter needs some iteration through the TM for learning, **it should set this variable in “initialize” method**. In the next subsections there are descriptions of mentioned methods.

3.3.1 Initialize

```

9     @abstractmethod
10    def initialize(self, source_language, target_language):
11        pass

```

The TMOP will call this method at the start of learning process (before the scans). Filters can initialize their variables (specially **num_of_scans**) and load models to the memory.

- **Important:** changing the value of **num_of_scans** after this method has no effect.

This method has two arguments “source_language” and “target_language”.

These two are **string variables** and they indicate the language of the source and target phrases in every translation unit. If the filter needs this information, it should store them.

For the **NewFilter**, the initialize method should indicate the number of scans over the TM. In this example it need **one pass** to calculate the mean and standard deviation of the ratios. So, the code for initialize method will be like code snippet 3.4.

Code Snippet 3.4: Example, initialize method

```
def initialize(self, source_language, target_language):
    self.num_of_scans = 1
    self.src_language = source_language
    self.trg_language = target_language

    self.n = 0.0
    self.sum = 0.0
    self.sum_sq = 0.0

    self.mean = 0.0
    self.var = 0.0
    return
```

The variable “n” is the number of TUs. The “sum” and “sum_sq” are the sum and sum of squared ratios. Finally, the “mean” and “var” are mean and standard deviation that are needed for decision making. This filter uses some mathematical functions for calculating mean and variance and it needs the “math” library. So, don’t forget to use “import math” at the start of the code in “NewFilter” file.

3.3.2 Process TU

```
17 | @abstractmethod
18 | def process_tu(self, tu, num_of_finished_scans):
19 |     pass
```

In every scan, for **each translation unit** this method is called. This method has two arguments “tu” and “num_of_finished_scans”. The “tu” is a TU class object (mentioned in section 3.2) and it contains the translation unit for processing. The other argument is “num_of_finished_scans”. It is an integer variable and it indicates the iteration number in learning phase. **It starts from “0”**.

For the **NewFilter**, this method should process the ratio between source and target phrase and add it to the sum variables. For making it simple and more readable, assume that the length of source and target phrase are **not zero**. The code for process_tu method is be like code snippet 3.5.

Code Snippet 3.5: Example, process_tu method

```
def process_tu(self, tu, num_of_finished_scans):
    ratio = len(tu.src_phrase) / len(tu.trg_phrase)

    self.n += 1
    self.sum += ratio
    self.sum_sq += ratio * ratio
```

3.3.3 Finalize a full scan

```
21 | @abstractmethod
22 | def do_after_a_full_scan(self, num_of_finished_scans):
23 |     pass
```

After each scan through the TM, this method is called. It could be used to finalize the calculation after each scan. This method has one argument “num_of_finished_scans”. It is an integer variable and it indicates the iteration number in learning phase. After the first scan the value of this argument is **1**.

3.3.4 Finalize

```
13 | @abstractmethod
14 | def finalize(self):
15 |     pass
```

This method is called at the end of learning process (after all of the scans). It is the last method to finalize the calculations and prepare everything for the decision phase.

In the **NewFilter**, this method should calculate the mean and standard deviation. It is also possible to these calculations in the “do_after_a_full_scan” method. The code for this method is be like code snippet 3.6.

Code Snippet 3.6: Example, finalize method

```
def finalize(self):
    if self.n <= 1:
        self.n = 2.0

    self.mean = self.sum / self.n
    tmp = (self.sum_sq - (self.sum * self.sum) / self.n)
    self.var = tmp / (self.n - 1)
    self.var = math.sqrt(self.var)
```

The first “if” is for preventing varinace from becoming infinite. For calculating the standard deviation in the right way, the filter needs two pass through the TM. In the first pass it should calculate the mean and in the second one calculate the standard deviation. In this example for making

the code less time-consuming, we used an approximation for the standard deviation.

3.3.5 Decide

```
25 | @abstractmethod
26 | def decide(self, tu):
27 |     return 'accept'
```

In the decision process this method is called for each translation unit. This method has one argument “tu” which is a TU class object. It should return a string from “accept”, “reject” or “neutral” based on the given TU.

In the **NewFilter**, the “decide” method should calculate the ratio for the TU and compare it with the mean and standard deviation. If the difference between the ratio and the mean is more than two standard deviations, it will reject the TU. The code for this method is be like code snippet 3.7.

Code Snippet 3.7: Example, decide method

```
def decide(self, tu):
    ratio = len(tu.src_phrase) / len(tu.trg_phrase)

    ratio -= self.mean
    ratio = math.fabs(ratio)

    if ratio <= 2 * self.var:
        return 'accept'
    return 'reject'
```

3.4 Writing a new policy manager

Writing a new policy is mostly like writing a new filter which is described in section 3.3. There is an abstract base class named “*abstract_policy.py*” and the code is shown in code snippet 3.8:

Code Snippet 3.8: Abstract policy manager class

```
1 | class AbstractPolicy:
2 |     __metaclass__ = ABCMeta
3 |
4 |     @abstractmethod
5 |     def __init__(self):
6 |         pass
7 |
8 |     @abstractmethod
9 |     def decide(self, result_list):
10 |         return 'accept'
```

There are two abstract methods for **initialization** and **decision**. All the policy managers **should derive this class** and override its methods. In the next subsections there are descriptions of mentioned methods.

3.4.1 Initialize

```
4 | @abstractmethod
5 | def __init__(self):
6 |     pass
```

This is the basic initialize method and it can contain some variable declaration and some calculations.

3.4.2 Decide

```
8 | @abstractmethod
9 | def decide(self, result_list):
10 |     return 'accept'
```

In the decision process this function is called for each translation unit. The input is named “*result_list*” and it is a list of answers from all the filters. The “*result_list*” is an array of tuples. Each tuple has two string values. The first one is the name of the filter and the second one is the answer of that filter. Here is a sample of “*result_list*”:

```
| result_list = [("filter1", "accept"), ("filter2", "reject")]
```

The *Policy Manager* should consider the filters names and their answers and come up with a decision.

The output is a string and it could be “**accept**”, “**reject**”, “neutral” or any other arbitrary string. For each output string a file is created by the TMOP and the TUs with that result are appended to that file.

As an example, the **OneNo** policy manager is described below. It counts the number of “reject” answers and if there are more than one rejects, it will return “reject” as the final decision.

Code Snippet 3.9: Example, OneNo policy manager

```
class OneNo(AbstractPolicy):
    def __init__(self):
        return

    def decide(self, result_list):
        tmp_arr = [1 for x in result_list if x[1] == "reject"]
        num_of_no_answers = len(tmp_arr)

        if num_of_no_answers > 0:
```

```
        return 'reject'  
    return 'accept'
```

Chapter 4

Implemented Filters

4.1 SampleFilter

Type: Language Independent, Sample

Description: This is just a sample filter to show the implementation of a simple filter. It only removes the translation units with empty source or target phrases.

Requirements: This filter only uses standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 0

Decision: It looks at the source and target phrases, and if either one of them was an empty string it rejects the TU.

4.2 LengthRatio & ReverseLengthRatio

Type: Language Independent, Variance, Standard Deviation, Ratio

Description: This filter checks the ratio of *the number of characters in the source phrase* to *the number of characters in the target phrase*. It calculates the average and variance of the ratio in the TM and rejects outliers. The reverse length ratio calculates the ratio of **target to source** length.

Requirements: This filter only uses standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 1

Decision: It calculates the ratio of source to target phrase length. If the difference between the ratio and the average is more than **2 standard deviations**, it will reject the TU.

4.3 WordRatio & ReverseWordRatio

Type: Language Independent, Regular Expression, Variance, Standard Deviation, Ratio, Word Count

Description: This filter checks the ratio of *the number of words in the source phrase* to *the number of words in the target phrase*. It calculates the average and variance of the ratio in the TM and rejects outliers. The reverse filter calculates everything for the reverse of the ratio. So, it checks the ratio of **target to source** word count. These filters use regular expressions for finding the number of words in the phrases. The regular expression used in the filters is written below:

```
"\w+|\$[\d\. ]+|\S+"
```

Requirements: This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 1

Decision: It calculates the ratio of source to target phrase length. If the difference between the ratio and the average is more than **2 standard deviations**, it will reject the TU.

4.4 RepeatedChars

Type: Language Independent, Regular Expression

Description: This filter searches for a sequence of 3 or more of the same character. It checks the number of these sequences on source and target side.

Requirements: This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 0

Decision: It checks the number of repeated characters sequences on source and target side. If these numbers are different, it will reject the TU.

4.5 RepeatedWords

Type: Language Independent, Regular Expression

Description: This filter searches for a sequence of 2 or more of the same word.

Requirements: This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 0

Decision: It checks both source and target phrases and if there are repeated words in a sequence, it will reject the TU.

4.6 WordLength

Type: Language Independent, Regular Expression, Variance, Standard Deviation

Description: This filter calculates the average length of words in source and target language. If there is a really big word in source or target phrase, it will reject the TU.

Requirements: This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 1

Decision: It checks the source words lengths with the average length of source words and the same thing for target words. If the difference between the word length and the average length is more than **3 standard deviations**, it will reject the TU.

4.7 TagFinder

Type: Language Independent, Regular Expression

Description: This filter searches for numbers, dates, email, URLs, XML tags, ref tags and image tags in the phrases.

Requirements: This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU.

Number of scans needed: 0

Decision: It checks both source and target phrases and if the number of these tags are different in source and target phrases, it will reject the TU.

4.8 Lang_Identifier

Type: Language Dependant

Description: This filter detects the language of source and target phrases and checks them with the languages indicated in config file.

Requirements: This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU and the source and target language in the config file.

Number of scans needed: 0

Decision: It detects both source and target phrases and if they are different from the languages in the config file, it will reject the TU.

4.9 AlignedProportion

Type: Language Independent, Word Alignment, Regular Expression

Description: This filter calculates the proportion of aligned words in the source and the target phrases. This proportion can show the relevance of two phrases in the TU.

Requirements: This filter needs alignment data as input and it **does not** calculate the word alignment itself. This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU. The words alignments should be provided in the TU.

Number of scans needed: 1

Decision: The filter checks the proportion of aligned words in the source and the target phrase separately and compares these proportions with the average proportion of the whole TM. If the difference between the TU proportions and the average ones are more than k standard deviations, it will reject the TU.

4.10 BigramAlignedProportion

Type: Language Independent, Word Alignment, Regular Expression

Description: This filter calculates the proportion of word bi-grams in the source and the target phrases. This proportion can show the relevance of two phrases in the TU.

Requirements: This filter needs alignment data as input and it **does not** calculate the word alignment itself. This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU. The words alignments should be provided in the TU.

Number of scans needed: 1

Decision: The filter checks the proportion of word bi-grams in the source and the target phrase separately and compares these proportions with the average proportion of the whole TM. If the difference between the TU proportions and the average ones are more than k standard deviations, it will reject the TU.

4.11 NumberOfUnalignedSequences

Type: Language Independent, Word Alignment, Regular Expression

Description: This filter checks the number of unaligned words sequences, normalized by the length of the phrase in the source and the target phrases. This ratio can help understanding the status of the source and the target phrases.

Requirements: This filter needs alignment data as input and it **does not** calculate the word alignment itself. This filter only uses Python standard

libraries and no additional library is needed.

Input: The only input needed for this filter is the TU. The words alignments should be provided in the TU.

Number of scans needed: 1

Decision: It calculates the number of unaligned words sequences, normalized by the length of the phrase in the source and the target phrase separately and compares these ratios with the average ratios of the whole TM. If the difference between the TU ratios and the average ones are more than k standard deviations, it will reject the TU.

4.12 LongestAlignedSequence & LongestUnalignedSequence

Type: Language Independent, Word Alignment, Regular Expression

Description: These filters check the length of the longest sequence of aligned/unaligned words, normalized by the length of the phrase in the source and the target phrases. This ratios can help understanding the relevance of the source and the target phrases.

Requirements: This filter needs alignment data as input and it **does not** calculate the word alignment itself. This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU. The words alignments should be provided in the TU.

Number of scans needed: 1

Decision: These filters calculate the length of the longest sequence of aligned/unaligned words, normalized by the length of the phrase in the source and the target phrase separately and compare these ratios with the average ratios of the whole TM. If the difference between the TU ratios and the average ones are more than k standard deviations, it will reject the TU.

4.13 AlignedSequenceLength & UnalignedSequenceLength

Type: Language Independent, Word Alignment, Regular Expression

Description: These filters calculate the average length of the sequences of

aligned/unaligned words in the source and the target phrases. It can help understanding the status of the source and the target phrases.

Requirements: This filter needs alignment data as input and it **does not** calculate the word alignment itself. This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU. The words alignments should be provided in the TU.

Number of scans needed: 1

Decision: These filters calculate the average length of the sequences of aligned/unaligned words in the source and the target phrases separately and compare these lengths with the average lengths of the whole TM. If the difference between the TU lengths and the average ones are more than k standard deviations, it will reject the TU.

4.14 FirstUnalignedWord & LastUnalignedWord

Type: Language Independent, Word Alignment, Regular Expression

Description: These filters calculate the position of the first/last unaligned word, normalized by the length of the phrase in the source and the target phrases. It can help understanding the status of the source and the target phrases.

Requirements: This filter needs alignment data as input and it **does not** calculate the word alignment itself. This filter only uses Python standard libraries and no additional library is needed.

Input: The only input needed for this filter is the TU. The words alignments should be provided in the TU.

Number of scans needed: 1

Decision: These filters calculate the position of the first/last unaligned word, normalized by the length of the phrase in the source and the target phrases separately and compare these ratios with the average ratios of the whole TM. If the difference between the TU ratios and the average ones are more than k standard deviations, it will reject the TU.

4.15 WE_Average & WE_Median

Type: Language Independent, Word Alignment, Word Embedding

Description: These filters create bilingual word embeddings for all words in the source and target segments. They calculate representative vectors for the source and the target segments by getting the average or median of the words in the segments. These vectors can represent the topic of the segments and the similarity between these vectors can show the relevance of the two segments.

Requirements: This filter uses standard libraries and the Gensim library.

Input: The only input needed for this filter is the TU.

Number of scans needed: 3

Decision: The filters create vector representations for the source and the target segments and compare them with cosine similarity. If the difference between this similarity and the average of all TUs is more than **2 standard deviations**, it will reject the TU.

4.16 WE_BestAlignScore

Type: Language Independent, Word Alignment, Word Embedding

Description: This filter creates bilingual word embeddings for all words in the source and target segments. It uses these vectors to align each source word with the most similar target word and vice versa. The cosine similarity of the vectors represents the score of each alignment and the average of these scores is used to measure the relevance of the two segments.

Requirements: This filter uses standard libraries and the Gensim library.

Input: The only input needed for this filter is the TU.

Number of scans needed: 3

Decision: The filter creates vector representations for the source and the target words and creates alignments between these words. It scores the TU with the average score of the alignments. If the difference between the score and the average of all TUs is more than **2 standard deviations**, it will reject the TU.

Chapter 5

Implemented Policy Managers

5.1 OneNo policy manager

Description:

This policy manager checks the results of all filters. If there is even one reject in the answers, it will reject the TU. Otherwise, the TU is accepted.

5.2 MajorityVoting policy manager

Description:

This policy manager checks the results of all filters. If at least fifty percent of the filters rejected the translation unit, it will reject the TU. Otherwise, the TU is accepted.