AutophrasePy: an Integrated Python Package for AutoPhrase

Qianyang Peng and Renxuan Wang
Computer Science Department, University of Illinois at Urbana-Champaign, IL, USA
{qp3, renxuan2}@illinois.edu

***Abstract*— AutoPhrase is a phrase mining tool which has been widely adopted in many mainstream applications. However, it has several limitations, such as not being user configurable and requiring independent execution of bash scripts. AutophrasePy is a programming project aiming at improving AutoPhrase. In this project, we mainly achieved two goals. First, we extended the language support of AutoPhrase, adding the functionality of Japanese and Arabic text training and segmentation. Second, we developed a Python package integrating all functions of AutoPhrase, which supports pip installation, automatic wikifiles downloading and customized model selection. We provided user-friendly APIs, which allows users to do phrase mining under any directory they like with just two simple Python functions.**

I. Introduction

Phrase mining is an important task in data mining. Compared to unigrams (single word), phrases are more meaningful and less ambiguous. For example, the word "United" could mean the United States, the United Nations or the United Airlines, while the phrase "United States" has a clear meaning. The general principle of phrase mining is fully exploiting information redundancy and data-driven criteria to determine phrase boundaries and salience.

Almost all state-of-the-art phrase mining methods require more or less phrase quality estimation and annotations from human experts. This would bring a huge limitation that they are not scalable to a new language, a new domain or genre. Besides, they may not fit domain-specific, dynamic, emerging applications like scientific domains, query logs, or social media.

To overcome this limitation, data-driven approaches are more and more applied. One novel automated phrase mining framework is AutoPhrase [1]. AutoPhrase utilizes high-quality phrases in KBs (e.g., Wiki) as positive phrase labels for distant training, and thus completely remove the human effort for labeling phrases.

Although AutoPhrase is a pathbreaking tool, it still has several limitations. First, it needs to be executed separately in command line. Second, it is not quite user configurable. The positions of training/testing files and the models are highly restricted, because the program looks for them in defined folders. Moreover, the wiki and stopwords for different languages should be downloaded manually by users. Although the code for crawling wiki is provided, it would be better to put the crawled wiki online and let users download them automatically when needed. Therefore, we developed AutophrasePy, an integrated Python package for AutoPhrase. We claim three main contributions during this project.

1) We fixed several faults in AutoPhrase, especially in its multilingual processing pipeline.
2) We extended the language support of AutoPhrase. It now supports Japanese and Arabic as well.
3) We developed a Python package as an integration and improvement of AutoPhrase. We name it AutophrasePy.

The rest of the paper is organized as follows. Section II thoroughly introduces how to use AutophrasePy, including installation and using it in Python. Section III gives an in-depth introduction of the background of phrase mining and AutoPhrase. Section IV illustrates our work of language support extension, while Section V states our effort for developing the Python package. In section VI we show the phrase mining results on a Japanese corpus and an Arabic corpus. We conclude the study in Section VII.

II. Usage of AutophrasePy

AutophrasePy is a downloadable python3 package from the Internet. It can be installed by pip3.

A. Installation

AutophrasePy has the following environment requirements:

- Operating system: Linux/MacOS
- Python 3.4 or above
- g++ supports compiling C++11
- Java Runtime Environment

You can either download the source code from our Github repository[1] and compile it yourself, or you can directly download the .tar.gz package and install it with *pip*.

Compilation:
python3 setup.py sdist
Installation:
pip3 install autophrase.tar.gz
You can also use pip to uninstall AutophrasePy:
pip3 uninstall autophrase

B. Execution

After installing AutophrasePy using *pip3*, user can use it like any other Python packages. There are two separated APIs provided for training and segmenting.

1) *train_model(args)*
   This is a function to train a segmentation model. The parameter list includes:
   - trainfile: path to the user provided training file.
   - language: language of the training file, need to be specified by users; currently can be English, Chinese, Arabic and Japanese; default is English.
   - pos_tagging: whether to do POS tagging during training; default is True.
   - thread: number of threads; default is 10.
   - min_sup: a hard threshold of raw frequency is specified for frequent phrase mining; default is 10.
2) *phrasal_segment(args)*
   This is a function to segment a file using a user specified model or pretrained model provided by us. The parameter list includes:
   - filename: path to the user provided segmenting file.
   - model_path: path to the user specified model. If you want to use your own segmentation model, you can pass the path in. Otherwise, just leave it empty.
   - pretrained_model: name of pretrained model. Currently, We provide DBLP and Yelp models for English, and one model for other three languages respectively. If you want to use your own model, just leave it empty.
   - language: language of the file to be segmented, need to be specified by users; default is English
   - pos_tagging: whether to do POS tagging; default is True
   - highlight_multi: threshold score of selecting a multi-word phrase; default is 0.5
   - highlight_single: threshold score of selecting a single-word phrase; default is 0.8
   - thread: number of threads; default is 10

When running either of the functions, the program will create an autophrase folder under the directory the user runs the Python script. Note that the user must have the permission to create folders there, otherwise the program will crash. The folder structure is:

```
autophrase/
        |—— data/
                |—— EN/
                |—— CN/
                |—— JA/
        |—— models/
        |—— tmp/
```

The data folder contains the wiki files and stopwords. Note that these files are language specific, thus when users specify the language, the program will first create the *data* folder, then create the language folder in it. The wiki files and stopwords for this language will be automatically downloaded and put into this language folder. The *models* folder is the place to store the training result (including the segmentation model, token mappings and scores of quality phrases). When running the *phrasal_segment* function, the *models* folder is the default folder the program searches for a segmentation model, and is also where we store a downloaded model when users choose to use a pretrained model. The *tmp* folder is the place to store some intermediate results, such as the tokenizing results and POS tagging results. One thing we should clarify is that the creation of folders and download of files are all based on the "do if not existing" pattern, which means any of these folders already exists will not be recreated or wiped out, except that the contents of the *tmp* folder will

[1]https://github.com/CS512-Autophrase-Demo/AutophrasePy

be removed every time before training/segmentation starts.

To make it more intuitive, we record a short demo video on YouTube[2]. We only show a segmentation example since the training usually takes roughly an hour.

III. About AutoPhrase

AutophrasePy is developed based on AutoPhrase. In this section, we give a comprehensive introduction of AutoPhrase.

A. Background and Related Work

A **phrase** is defined as a sequence of words that appear consecutively in the text, forming a complete semantic unit in certain contexts of the given documents [2]. The **phrase quality** is defined to be the probability of a word sequence being a complete semantic unit, meeting the four criteria: **Popularity**, **Concordance**, **Informativeness** and **Completeness**[3]. Phrase mining is the process of automatic extraction of high-quality phrases in a given corpus. Representing the text with *quality phrases* instead of *n-grams* can improve computational models for applications such as information extraction/retrieval, taxonomy construction, and topic modeling.

Unfortunately, almost all other state-of-the-art methods require human experts at certain levels. Most of them [4], [5], [6] rely on an underlying linguistic analyzers to help locate phrase mentions. These analyzers are often complex and need training, like dependency parsers. SegPhrase [3] is proposed by the same research group as AutoPhrase. It outperforms many other approaches [4], [5], [6], [7], [8], [9], but still needs nontrivial human effort. Domain experts are needed to first carefully select hundreds of varying-quality phrases from millions of candidates, and then annotate them with binary labels. The dependency on these analyzers, domain-dependent language rules and costly human labeling work makes it challenging to extend these approaches to text corpora of new domains and genres.

To overcome this limitation, data-driven approaches are recently instead used to make use of frequency statistics in the corpus to address both candidate generation and quality estimation [7], [8], [9], [3]. They do not rely on complex linguistic feature generation, domain-specific rules or extensive labeling efforts. Instead, they rely on large corpora containing hundreds of thousands of documents to help deliver superior performance. It is worth mentioning that all these approaches still require human experts for designing rules or labeling phrases. Therefore, extending them to work automatically is challenging.

[2]https://www.youtube.com/watch?v=NQtu15yKgM4

B. The Key Idea Formation of AutoPhrase

The key component of AutoPhrase is called *robust positive-only distant training*. The first phase is establishing the set of phrase candidates that contains all n-grams over a minimum threshold τ (typically 30) in the corpus. The basic belief is that this set is huge and the majority of the phrases are actually of inferior quality. In practice, among millions of phrase candidates, only about 10% are in good quality. The way of labeling words is using public knowledge bases like Wikipedia. The knowledge bases usually encode a considerable number of high-quality phrases in the titles, keywords, etc.

The quality phrases from knowledge bases are placed in a positive pool, while the phrase candidates derived from the given corpus but fail to match any high-quality phrase form a large but noisy negative pool. Then, the most important step is to independently train T unpruned decision trees, and they together serve as an ensemble classifier (random forest) that averages the results of T base classifiers. As shown in Figure 1, for each decision tree, K ($K = 100$) phrase candidates are draw with replacement from the positive pool and the negative pool respectively. The ideal error of a decision tree trained on this size-2K perturbed training set is $\frac{\sigma}{2K} \approx 10\%$, which approximately equals to the proportion of switched labels among all phrase candidates. The error rate of the random forest is thus

$$\text{ensemble\_error}(T) = \sum_{t=\lfloor 1+T/2 \rfloor}^{T} \binom{T}{t} p^t (1-p)^{T-t}$$

where $p = 10\%$. We can easily observe that the ensemble error is approaching 0 when T grows.

Another new technique proposed in AutoPhrase is *POS-guided phrasal segmentation*. Phrasal segmentation addresses the challenge of measuring *completeness* by locating all phrase mentions in the corpus and rectifying their frequencies obtained originally via string matching. Given POS tags, we can have *POS quality score* $T(t_{[l,r)} = p(\lceil w_l \dots w_r) | t)$ besides the phrase quality score $Q(w_1 w_2 \dots w_n) =$

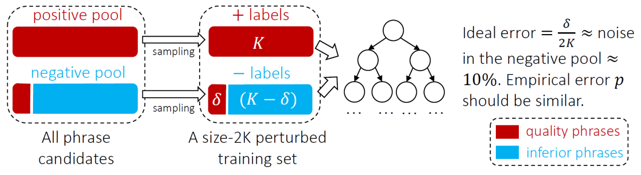
positive pool
negative pool
sampling
sampling
All phrase candidates
+ labels
K
− labels
δ
$(K-\delta)$
A size-2K perturbed training set
Ideal error = $\frac{\delta}{2K}$ ≈ noise in the negative pool ≈ 10%. Empirical error p should be similar.
quality phrases
inferior phrases

Fig. 1. The illustration of each base classifier in AutoPhrase [1]

$p(\lceil w_1 w_2 \dots w_n \rfloor | w_1 w_2 \dots w_n)$, where $\langle w_i, t_i \rangle$ refers to a pair of words and its POS tag. The shallow syntactic information in POS tags can guide the phrasal segmentation model locating the boundaries of phrases more accurately.

C. Why Autophrase Could Support Multiple Languages

There are three main reasons about why Autophrase could support multiple languages:

- Autophrase does not require a language expert to manually label any feature in the corpus. Instead, all it needs is the wiki entry titles of that specific language which could be automatically generated.
- In the preprocessing step Autophrase requires the POS tags of the document, however it does not care about what exactly the tagset is. Thus any tagset in any language could fit in the Autophrase model well.
- The algorithm of Autophrase is totally frequency statistics based and does not rely on the language grammar.

IV. Language Support Extension

The pipeline of language extension is shown in Figure 2. For the four phases of AutoPhrase, we need to extend three of them. The work need to be done in each phase is shown in red boxes.

A. Quality Phrase Extraction

The first work of adding support for a new language is to extract quality wiki phrases in that language. AutophrasePy used an 2-phase algorithm to derive high quality phrases of any specific language using the language specified wiki dataset. The extracted high quality phrases are used as the knowledge base of the Segphrase step of Autophrase. The first phase of the algorithm is the entity extraction, in this phase all the available entities of a specified language is extracted from wiki database. The second phase is a high quality phrase filter, which can

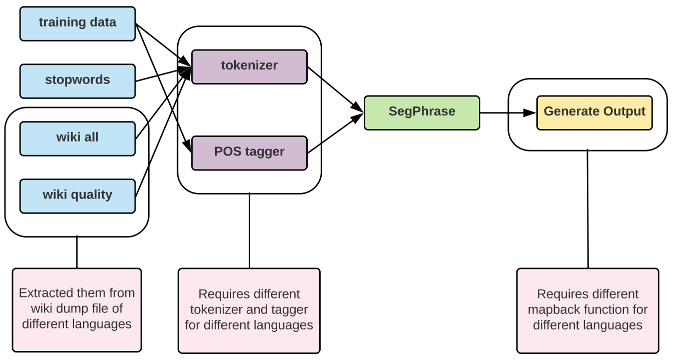
training data
stopwords
wiki all
wiki quality
tokenizer
POS tagger
SegPhrase
Generate Output
Extracted them from wiki dump file of different languages
Requires different tokenizer and tagger for different languages
Requires different mapback function for different languages

Fig. 2. AutoPhrase Language Extension Pipeline

distinguish high quality phrases and put them in a separate file.

1) Entity Extraction: In our algorithm the candidate set is extracted from the wiki data dump file, which contains all the wiki page articles of a specified language. Erich Schubert[10] proposed a efficient algorithm to extract wiki entities and evaluate their qualities.

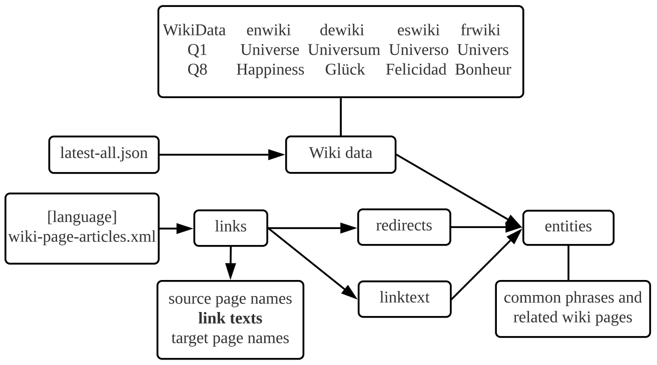

Fig. 3. Wiki Phrase Extraction Pipeline

The intuitive method to extract quality phrases from wiki data is to crawl a list of all wiki page titles. Until May 2018, there are 5,646,151 articles in the English wiki. This intuitive method is able to derive a list of phrases, however it also has two drawbacks:

- The title of a phrase could define an entity but the same entity could be represented by different phrases. For example, the entity "Law of the United States" is also known as "Federal law" or "Federal statute". However, the title for the wiki page is "Law of the United States", and the aliases of this entity will not be considered as an phrase.
- Many of the entities are not of high quality phrases. For example, a lot of natural numbers

has its own wiki column but we do not want to consider them as high quality. Also, we want to distinguish the more reliable entities from the less reliable ones, thus deriving a smaller set of high quality phrases.

Erich's algorithm is a simple approach for "learning" a corpus of named entities. It contains two phases. In the first pass, he parses a complete Wikipedia dump of a specific language. From each regular article page he extracts:

1) the redirect target, if the article is a redirect.
2) all links, except in Wikipedia templates and references.
3) all text used for linking to other articles.
4) full text of article for search.

In the second pass, he reads all the redirects and page links, then iterate over the known link texts. For every link text, he queries the Lucene database for the pages where this phrase occurs.

Take the phrase "obamacare" for example. There are 251 articles in Wikipedia that used this phrase. An article, Ralph Hudgens, links to Obamacare, which is a redirect to Patient Protection and Affordable Care Act. Thus, we count the article Ralph Hudgens supporting Obamacare is an entity.

There are some language specific pitfalls in this algorithm, that is a text link could mean different meaning in different language's wiki page. For example, "bayern" in German means Bavaria, but in English wikipedia it usually refers to the soccer club FC Bayern. It will somehow influence the classification of an entity. To address this issue, in AutophrasePy we only process the wikidump in one specific language at one time to reduce the entity conflict in different languages.

2) High Quality Phrase Filter: Our high quality filter algorithm inherits the algorithm of Jingbo Shang[11], but did some modification to better support language extension. Generally the quality phrase set is a subset of the overall phrase set, but need to satisfy the requirements below:

1) The first word in the phrase is not a stop word.
2) The count of stop words is no more than $\frac{1}{3}$ of the total number of words in the phrase.
3) There is no separator (e.g. ',','.',';') in the phrase.
4) The score of a phrase, which is the frequency of exact link text occurred + the number of articles that linked to the article, or the support of a phrase, which is number of articles the text was found in, is higher than their corresponding threshold.

The reason why we do not want any separator in the phrase is because punctuation is not tokenized in out algorithm, thus phrase with separator in it will never be matched with the tokenized text. The overall idea of the algorithm is similar to PageRank, which means a high quality phrase should be well formed and with enough entity to support it.

B. Adding Tokenizer and POS Tagger

The previous subsection is about extension in the first phase. After getting knowledge base (wiki) and stopwords for a new language, we need to add tokenizer and POS tagger in the second phase.

In our work, adding support for Japanese is relatively easy, because Japanese tokenizer and POS tagger is not difficult to integrate. However, for Arabic, things are more difficult due to three reasons:

1) The current POS tagger used in AutoPhrase is treetagger [12], which does not support Arabic. Very few other tokenizers/taggers support Arabic either.
2) Arabic text goes from right to left. This is not a severe problem as Arabic can be UTF-8 encoded, so that the program could handle it. But when mapping back, if we still use *<phrase>* pairs to highlight the quality phrases in segmentation results, it will display like a mess that some text goes from left to right while others goes from right to left, and the cursor will behave funnily that "jump" forwards and backwards when you try to move it with keyboard.
3) Open-source, large Arabic corpus are hard to find. We found a survey of freely available Arabic corpora [13], but unluckily, most of the corpora mentioned are no longer valid.

To address the first challenge, we carried a comprehensive survey. We found that most popular NLP tools do not support Arabic, such as treetagger, Apache OpenNLP[3], NLTK[4] and spaCy[5]. Most Arabic NLP tools are only described in papers or demo-oriented [14], [15], [16], which cannot be used as a part of our task. Luckily, we found that the Stanford POS tagger [17] not only supports Arabic, but also

[3]https://opennlp.apache.org/
[4]https://www.nltk.org/
[5]https://spacy.io/

is an open-source project written in Java, thus can be inserted in our tokenizer source code, which is written in Java as well.

In general, there are two kinds of text preprocessing pipelines in AutoPhrase. As shown in Figure 4,for English-like languages such as English, French, Spanish, the text will go through a Lucene Analyzer [6] to get tokens, and also go through the treetagger to get POS tags. Whereas for more complicated languages like Chinese, Japanese and Arabic which treetagger does not support, AutoPhrase provides an interface names *SpecialTagger*, where we can implement it with any third-party taggers. The SpecialTagger should take the raw text and return tokens and tags all at once, as shown in Figure 5. In AutoPrase, Chinese is handled by ansj_seg[7], which is a tool developed by NLPChina. In our work, we use Stanford POS tagger to implement the SpecialTagger interface for Arabic.

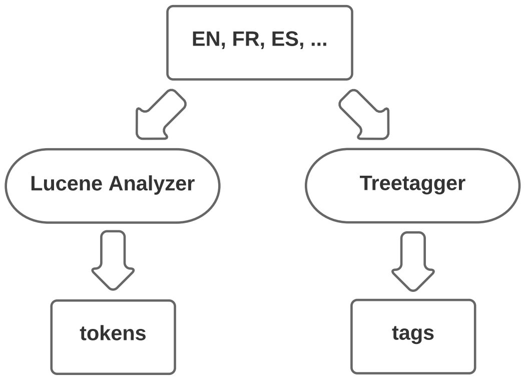

Fig. 4. Pipeline of English-like Languages

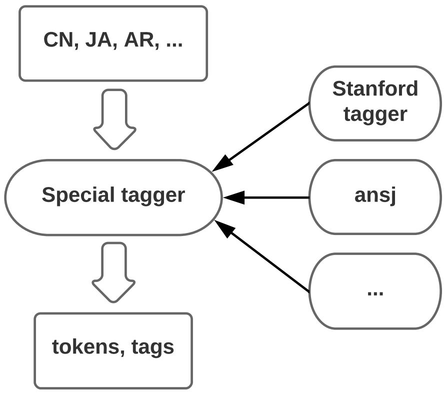

Fig. 5. Pipeline of Languages not Supported by Treetagger

[6]http://www.apache.org/dyn/closer.lua/lucene/java/6.6.3
[7]https://github.com/NLPchina/ansj_seg

As for the second challenge, we use the Arabic translation of the word "phrase" to highlight the quality phrases, so that the result will all display from right to left.

V. Python Package Implement

This section is the key part about how we implement AutophrasePy.

A. Python Package Installer

The Python Packaging Authority (PyPA), which is a working group that maintains many of the relevant projects in Python packaging, recommends developers to use pip for package management and distribution. Pip is a package management system used to install and manage software packages written in Python. Many packages can be found in the default source for packages and their dependencies — Python Package Index(PyPI). PyPI is the official third-party software repository for Python.

The installer of AutophrasePy is implemented and configured following the guideline of PyPA. The installer mainly contains four files:

- setup.py: This file contains the main installation logic, including code compilation, package configuration, file distribution, and version related information. It also contains the package compatibility information, including the operating system and python version supported by this package.
- setup.cfg: This is the configuration file containing the command line specifications needed when installing the package and the non-default options of the package plugins. In AutophrasePy we configure the style of our package version number and the paths to versioneer files.
- versioneer.py: This is a version string management plugin which could automatically generate formated version string for the distributed python package. There are several styles to choose from and what we choose is pep440.
- MANIFEST.in: This is a configuration file of what files in the package need to be put into the installation. Notice that this file can only influence the files located in the package but cannot influence files not originally in the package, for example the binary files generated during compilation.

B. Rewrite AutoPhrase in Python

AutoPhrase has several parts and written in different programming languages. The tokenizer is written in Java, the treetagger is a integrated tool whose core part is written in Perl and can be executed by bash, the SegPhrase part is written in C++, and the mapping back part is again written in Java. These parts make up a whole pipeline and are driven by bash scripts.

To rewrite AutoPhrase in Python, we create a Python class, and each original part of AutoPhrase becomes a member function. These member funtions, with some helper functions, are not provided to users. For users, we export two static functions mentioned in Section II as APIs so that they do not need to create an object of this class. These two functions will call the necessary functions inside.

Notice that the Python package is installed in some root directory (like */usr/lib*), where the program usually has no permission to create or modify files/folders. Thus, we change the original logic in AutoPhrase. We put all folders generated during the usage of AutophrasePy to the directory where the Python script is running. This results in modification of the C++ code, which originally statically restrict the path of folders.

C. Searching for Substitute of Treetagger

AutoPhrase uses treetagger for POS tagging. Treetagger is written in Perl, the developers provide shell scripts, which is the only way to use. This is quite inconvenient in several ways:

1) When we want to call treetagger in Python, we have to use *os.system()* command, which is actually not really using treetagger by Python but relying on the Linux system command.
2) Using treetagger in Windows is notoriously annoying. You will need to put treetagger under the root directory of drive C:\ and modify the PATH environment variable.
3) The parameter files need to be downloaded manually. Moreover, you have to put it under certain folder and rename it.

Therefore, we want to look for another tool to substitute treetagger. Consider the tokenizer and POS tagger as a whole system, we need the tool to have the listed advantages below:

- The tool must be robust. It should currently be an active project under regular maintenance. It should not have any severe unsettled open issue such as failure due to the input containing non-ascii character or the size of input is too big.
- The tool must be fast. It processing speed shouldn't be significantly slower than 100Mb/min. This requires the tool to be efficiently implemented with a fine management of time complexity and memory cache.
- The tool must be accurate. As tokenizing and tagging is two initial steps in the pipeline, a low accuracy in any of these step will directly effects the final accuracy.
- The tool should support large input. It should not consume significantly more computing resource than it should when the input size is larger than 1 GB. We need large training dataset to improve the performance of our whole system.
- The tool must support as many natural languages as possible.

We carried a survey and compared several most popular NLP tools up to date.

1) Stanford NLP: Stanford NLP tools are originally implemented in Java, and is extended by other developers to Docker, .NET, GATE, Go, Javascript, Matlab, PHP, Python and Ruby. It supports Arabic, Chinese, English, French, German and Spanish. The problem is it is too slow for large input. For our English training file, it takes about 10 minutes to finish tokenization. Besides, we cannot turn off its warnings, which becomes annoying when encounters untokenizable characters in the raw text.

2) TreeTagger: Treetagger is written in Perl and can only be used by shell scripts. It supports Bulgarian, Catalan, Chinese, Coptic, Czech, Danish, Dutch, English, Estonian, Finnish, French, Galician, Greek, German, Italian, Korean, Latin, Mongolian, Polish, Portuguese, Romanian, Russian, Slovak, Slovenian, Spanish and Swahili. This is the a very tool and supports most languages.

3) Apache OpenNLP: It provides Java packages and supports Danish, German, English, Spanish, Dutch, Portuguese and Swedish.

4) NLTK: It is implemented in Python. Supported languages: Chinese, English, Hindi, Portuguese, Catalan, Spanish, Dutch. Its drawback is it requires too many dependent packages need to be downloaded. The installation can be very time consuming.

5) SpaCy: It is implemented in Python, and declared to be the fastest POS tagger in the world.

The algorithm of SpaCy's POS tagger is a greedy decoding with the averaged perceptron. Supported languages include English, German, Spanish, Portuguese, French, Italian and Dutch. Unfortunately, it does not return token mappings, which is required in our SegPhrase part.

We can see that there is no perfect tool which can be both fast and supportive for all the languages we need. As a result, we decided to keep treeTagger due to its efficiency and rich language support, and use Stanford NLP tool only when necessary.

VI. Experimantal Result

A. Japanese Corpus

Score	Phrase	Translation
0.9716255566	ブラック アウト	blackout
0.9714860494	ファースト クラス	first class
0.9714703448	**銃撃 戦**	gunfight
0.9713316479	**構成 要素**	element
0.971241847	地方 検事	district attorney
0.9710232895	アップ ロード	upload
0.9709262931	**フロント ガラス**	front glass
0.9707894927	オープン ソース	open source
0.9706795211	**ダイレクト アタック**	direct attack
0.9704321159	ニセ モノ	fake things
0.9704189407	うずまき ナルト	uzumaki naruto

Fig. 6. The results of AutophrasePy on the Japanese Corpus with translations

The results in Figure 6 is the top score items in the result trained with a Japanese video subtitles dataset. The Bold phrases are the phrases not originally existed in the knowledge base but derived from our algorithm. In this example all these derived phrases are of very high quality, as although they are not in our knowledge base, they are all phrases recorded by the Japanese phrase dictionary.

B. Arabic Corpus

As mentioned in Section IV-B, it is very difficult to find open-source, large Arabic corpus. For the occasion, we use a BBC News UK Corpus written in Arabic, simply to test the functionality of AutophrasePy. The results are shown in Figure 7.

It is obvious that the quality phrases are limited in the field of the news related to UK, US and Arabia, and the phrases are not getting high scores. This is because the corpus we use is quite small and the phrases cannot have a high frequency. We believe we will get better result when we are equipped with larger corpus.

Score	Phrase	Translation
0.8838333333	مجلس الأمن	Security Council
0.8155940529	الأمريكي	American
0.8123253435	دارفور	Darfur
0.8074543106	المصري	Egyptian
0.8071304090	البريطانية	British
0.8047345504	لندن	London
0.8002076245	الأمريكية	America
0.7984461954	العراق	Iraq
0.7955621712	صدام	Saddam
0.7893333333	صفحة الاخبار الاخبار العالمية هيئة الاذاعة	World News
0.7564212951	بوش	Bush

Fig. 7. The results of AutophrasePy on the Arabic Corpus with translations

VII. Conclusion

In this paper, we describe a Python package for AutoPhrase, namely, AutophrasePy. We introduce the background of phrase mining and the limitation of current expert-based approaches. We give a comprehensive analysis of AutoPhrase, including how it removes the human effort for labeling phrases and how it supports multiple languages. We also explain the necessity of developing this Python package. We illustrate the usage and implementation of AutophrasePy in a very detailed way, including its installation and using in Python. In AutophrasePy, we extend the language support so that it now includes Japanese and Arabic. We conduct two case studies, one on each language, and the results are promising.

For future work, we can submit the package to Python Package Authority, and provide more wiki files, pre-trained segmentation models and APIs to make AutophrasePy more powerful.

Acknowledgement

We would like to express our sincere gratitude to the original author of AutoPhrase, Jingbo Shang, who gives a lot of support and help during our project. We also want to thank Professor Jiawei Han, not only for providing this opportunity so that we could gain a deeper understanding of data mining, but also for his excellent teaching during the whole semester.

References

[1] Jingbo Shang, Jialu Liu, Meng Jiang, Xiang Ren, Clare R Voss, and Jiawei Han. Automated phrase mining from massive text corpora. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[2] Geoffrey Finch. Linguistic terms and concepts. *New York, NY: St. Martin's*, 2000.

[3] Jialu Liu, Jingbo Shang, Chi Wang, Xiang Ren, and Jiawei Han. Mining quality phrases from massive text corpora. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1729–1744. ACM, 2015.

[4] Katerina Frantzi, Sophia Ananiadou, and Hideki Mima. Automatic recognition of multi-word terms:. the c-value/nc-value method. *International journal on digital libraries*, 3(2):115–130, 2000.

[5] Youngja Park, Roy J Byrd, and Branimir K Boguraev. Automatic glossary extraction: beyond terminology identification. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics, 2002.

[6] Ziqi Zhang, José Iria, Christopher Brewster, and Fabio Ciravegna. A comparative evaluation of term recognition algorithms. 2008.

[7] Paul Deane. A nonparametric method for extraction of candidate phrasal terms. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 605–613. Association for Computational Linguistics, 2005.

[8] Ahmed El-Kishky, Yanglei Song, Chi Wang, Clare R Voss, and Jiawei Han. Scalable topical phrase mining from text corpora. *Proceedings of the VLDB Endowment*, 8(3):305–316, 2014.

[9] Aditya Parameswaran, Hector Garcia-Molina, and Anand Rajaraman. Towards the web of concepts: Extracting concepts from large datasets. *Proceedings of the VLDB Endowment*, 3(1-2):566–577, 2010.

[10] Erich Schubert. Wikipedia entities and synonyms, 2016.

[11] Jingbo Shang. High quality phrase filter, 2017.

[12] Helmut Schmid. Improvements in part-of-speech tagging with an application to german. In *In proceedings of the acl sigdat-workshop*. Citeseer, 1995.

[13] Wajdi Zaghouani. Critical survey of the freely available arabic corpora. *arXiv preprint arXiv:1702.07835*, 2017.

[14] Ahmed Abdelali, Kareem Darwish, Nadir Durrani, and Hamdy Mubarak. Farasa: A fast and furious segmenter for arabic. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 11–16, 2016.

[15] Arfath Pasha, Mohamed Al-Badrashiny, Mona T Diab, Ahmed El Kholy, Ramy Eskander, Nizar Habash, Manoj Pooleery, Owen Rambow, and Ryan Roth. Madamira: A fast, comprehensive tool for morphological analysis and disambiguation of arabic. In *LREC*, volume 14, pages 1094–1101, 2014.

[16] Kareem Darwish, Ahmed Abdelali, and Hamdy Mubarak. Using stem-templates to improve arabic pos and gender/number tagging. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, may 2014. European Language Resources Association (ELRA).

[17] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.