The LEXenstein Manual

Gustavo Henrique Paetzold

May 2015

Table of contents

Chapter 1

Introduction

LEXenstein is a tool for the benchmarking of Lexical Simplification systems. It contains several methods, tools and resources for one to easily create Lexical Simplification systems and test them in various distinct ways. It takes Lexical Simplification as a process that can be modeled as a pipeline of sub-processes, as illustrated in Figure 1.1.

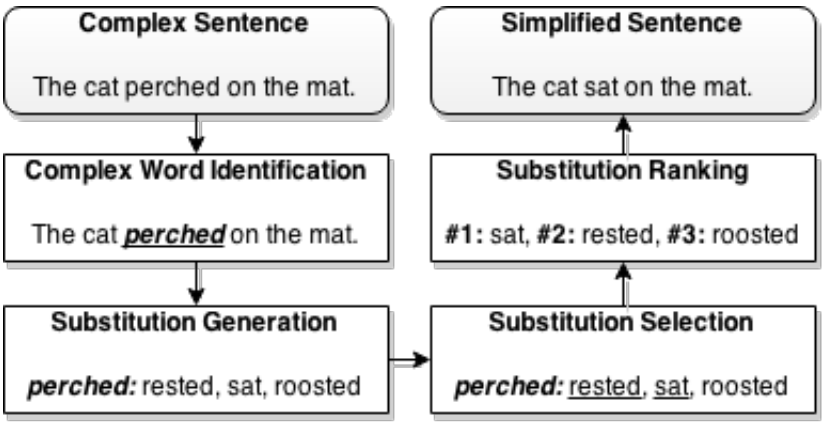

Fig. 1.1 Lexical Simplification pipeline

Each task in Figure 1.1 can be described as:

1. **Complex Word Identification**: Task of deciding which words of a given sentence may not be understood by a given target audience and hence must be simplified.
2. **Substitution Generation**: Task of finding pairs of words or expressions that share the same meaning and are interchangeable in some context.
3. **Substitution Selection**: Task of deciding which is the meaning of a given complex ambiguous word in a sentence to be simplified, and then selecting which substitutions available also represent that meaning.

4. **Substitution Ranking**: Task of ranking the remaining substitutions of a given complex word by their simplicity.

Its name is a reference to the *Frankenstein* novel, published by [28]. It refers to the fact that one can, as Victor Frankenstein did, create an entire Lexical Simplification "creature" by combining multiple "pieces" of the Lexical Simplification pipeline together.

LEXenstein is divided in several modules: Complex Word Identification, Substitution Generation, Substitution Selection, Substitution Ranking, Feature Estimation, Evaluation, Text Adorning and Spelling Correction. In the next Sections, we describe the VICTOR and CWICTOR file formats (Section 3.1), explain how to properly setup LEXenstein (Section 2), and also describe in detail the components included in each module of the LEXenstein tool (Section 4).

Chapter 2

Installation

LEXenstein is a library written entirely in Python. It hence requires for Python 2.7.* to be installed in the user's machine. We have not yet tried to run LEXenstein over Python 3.*. To install LEXenstein, please follow the steps below:

1. Download the tool from **omitted for review**.

2. Copy the "lexenstein" folder included in the package into the folder of your project.

3. Access LEXenstein by importing its modules in the following fashion:

```
from lexenstein.morphadorner import *
from lexenstein.spelling import *
from lexenstein.features import *
from lexenstein.identifiers import *
from lexenstein.generators import *
from lexenstein.selectors import *
from lexenstein.rankers import *
from lexenstein.evaluators import *
from lexenstein.util import *
```

In the near future, we will allow for one to install LEXenstein by using standard installation routines, such as through a "setup.py" file, or pip. LEXenstein requires for several libraries and toolkits to be included in the user's Python 2.7.* installation. The following Sections explain which libraries and toolkits are required, where to get them and how to install them.

2.1 Required Libraries

2.1.1 Morph Adorner Toolkit

The Morph Adorner Toolkit [21] is a set of Java applications that facilitate the access to Morph Adorner's functionalities [7]. This tool is used by LEXenstein's Substitution Generation module to create inflections for generated substitutions. To install it, follow the steps below:

1. Download the tool from http://ghpaetzold.github.io/MorphAdornerToolkit/
2. Place it in a folder of your choice

Since the tool does not require any compilation, all you need to do is use the path in which you installed it to create instances of the **MorphAdornerToolkit** class, which can be found in LEXenstein's Morph Adorning module.

2.1.2 NLTK

NLTK [3] is a set of resources and algorithms for tasks related, but not restricted to, Natural Language Processing. To install it, please follow the steps provided in: http://www.nltk.org/install.html. Once you have NLTK installed in your Python distribution, please download all additional resources available by following this tutorial: http://www.nltk.org/data.html.

2.1.3 KenLM

KenLM [12] is a tool for fast language model creation and querying. LEXenstein's modules use KenLM to access the data in binary language models for various tasks, such as feature calculation and substitution filtering. To install it, please follow the steps below:

1. Download or clone KenLM from https://github.com/kpu/kenlm
2. Place it in a folder of your choice
3. Navigate to the installation folder in a terminal and run: **python setup.py install**

If no problems occur, KenLM should now be installed in your Python distribution. To verify whether or not the installation was successful, open Python and try importing the library with the following line of code: **import kenlm**. If no errors occur, then the installation was successful.

2.1.4 Scipy and Numpy

Scipy and Numpy [16] are tools that offer great utility for projects and applications in the fields of mathematics, science, and engineering. To install them, please follow the instructions in: http://www.scipy.org/install.html.

2.1.5 Gensim

Gensim [24] is a set of algorithms for unsupervised semantic modeling. LEXenstein uses Gensim to read word vector models. To install it, follow the instructions in: https://radimrehurek.com/gensim/install.html.

2.1.6 PyWSD

PyWSD [30] is a library that offers access to several Word Sense Disambiguation algorithms. LEXenstein's uses this library to filter substitutions. To install it, follow the steps below:

1. Download or clone PyWSD from https://github.com/alvations/pywsd
2. Place it in a folder of your choice
3. Navigate to the installation folder in a terminal and run: **python setup.py install**

If no problems occur, PyWSD should now be installed in your Python distribution. To verify whether or not the installation was successful, open Python and try importing the library with the following line of code: **import pywsd**. If no errors occur, then the installation was successful.

2.1.7 Scikit-Learn

Scikit-Learn [23] is a set of tools for data mining, data analysis and machine learning. LEXenstein uses this library to learn ranking models. To install it, follow the instructions in: http://scikit-learn.org/stable/install.html.

2.1.8 SVM-Rank

SVM-Rank [15] is a tool that allows for one to use Support Vector Machines in ranking setups. LEXenstein uses this library to learn ranking models. To install it, follow the instructions in: http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html.

2.1.9 Stanford Tagger

The Stanford Tagger [17] is a tool that allows for one to annotate sentences with Part-of-Speech (POS) tags. LEXenstein uses this library to find the POS tag of a target word in a sentence. To install it, download the application's latest version from http://nlp.stanford.edu/software/tagger.shtml. Inside the package, you will be able to find the **stanford-tagger.jar** executable and pre-trained tagging models inside the **/models/** folder required by some of LEXenstein's substitution generators.

2.1.10 Other Libraries

LEXenstein also uses various other well-known Python modules, they are: xml, re, urllib2, subprocess, codecs and os.

Chapter 3

The VICTOR and CWICTOR Formats

In order to standardize input and output within the LEXenstein framework, we have conceived the VICTOR and CWICTOR formats: an elegant way of representing data for both the training and testing of Lexical Simplification models and systems. It is a reference to Victor Frankenstein, the main character in the Frankenstein novel [28], and creator of the ever so popular Frankenstein's Monster.

3.1 The VICTOR Format

The VICTOR format was conceived to represent datasets for the tasks of Substitution Generation, Selection and Ranking. Each line of a file in VICTOR format is structured as illustrated in Example 3.1, where S_i is the ith sentence in the dataset, w_i a target complex word in the h_ith position of S_i, c_i^j a substitution candidate and r_i^j its simplicity ranking.

$$\langle \mathrm{S}_i \rangle \ \langle \mathrm{w}_i \rangle \ \langle \mathrm{h}_i \rangle \ \left\langle \mathrm{r}_i^1 : \mathrm{c}_i^1 \right\rangle \left\langle \mathrm{r}_i^2 : \mathrm{c}_i^2 \right\rangle \cdots \left\langle \mathrm{r}_i^{n-1} : \mathrm{c}_i^{n-1} \right\rangle, \langle \mathrm{r}_i^n : \mathrm{c}_i^n \rangle \tag{3.1}$$

Each bracketed component in the example above is separated by a tabulation marker. Examples of files with such notation can be found in "resources/victor_datasets".

3.2 The CWICTOR Format

The CWICTOR format was conceived to represent datasets for the task of Complex Word Identification. Each line of a file in CWICTOR format is structured as illustrated in Exam-

ple 3.2, where S_i is the ith sentence in the dataset, w_i a the word in the h_ith position of S_i, and l_i a binary label, which must have value 1 if w_i is complex, and value 0 otherwise.

$$\langle S_i \rangle \ \langle w_i \rangle \ \langle h_i \rangle \ \langle l_i \rangle \tag{3.2}$$

Each bracketed component in the example above is separated by a tabulation marker. Examples of files with such notation can be found in “resources/cwictor_datasets”.

Chapter 4

Modules

4.1 Spelling Correction

LEXenstein's Spelling Correction module allows for one to correct misspelled words. The module is used by all generators in order to ensure that words and lemmas don't have their grammaticality compromised during inflection. It includes the NorvigCorrector class, which employs the spelling correction algorithm of Norvig[1].

```
from lexenstein.spelling import *

nc = NorvigCorrector('corpus.txt')
print(nc.correct('mystake'))
print(nc.correct('speling'))
print(nc.correct('beatiful'))
```

The output produced by the script above will be:

```
mistake
spelling
beautiful
```

4.2 Text Adorning

LEXenstein's Text Adorning module provides a Python interface to the Morph Adorner Toolkit [21], a set of Java tools that facilitates the access to Morph Adorner's functionalities.

[1] http://norvig.com/spell-correct.html

The class MorphAdornerToolkit provides easy access to word lemmatization, word stemming, syllable splitting, noun inflection, verb tensing, verb conjugation and adjective/adverb inflection. Below is an example of how to create and use a MorphAdornerToolkit object:

```
from lexenstein.morphadorner import MorphAdornerToolkit

m = MorphAdornerToolkit('./morph/')

lemmas = m.lemmatizeWords(['doing', 'geese'])
print('Lemmas:')
print(str(lemmas)+'\n')

stems = m.stemWords(['doing', 'geese'])
print('Stems:')
print(str(stems)+'\n')

tenses = m.tenseVerbs(['do'], ['doing'])
print('Tenses:')
print(str(tenses)+'\n')

verbs = m.conjugateVerbs(['do', 'sit'], 'PRESENT_PARTICIPLE')
print('Verbs:')
print(str(verbs)+'\n')

nouns = m.inflectNouns(['goose', 'chair'], 'plural')
print('Nouns:')
print(str(nouns)+'\n')

syllables = m.splitSyllables(['persevere', 'sitting'])
print('Syllables:')
print(str(syllables)+'\n')

adjectives = m.inflectAdjectives(['nice', 'pretty'], 'comparative')
print('Adjectives:')
print(str(adjectives)+'\n')
```

The output produced by the script above will be:

```
Lemmas:
['do', 'goose']
```

```
Stems:
['do', 'gees']

Tenses:
['PRESENT_PARTICIPLE']

Verbs:
['doing', 'sitting']

Nouns:
['geese', 'chairs']

Syllables:
['per-se-vere', 'sit-ting']

Adjectives:
['nicer', 'prettier']
```

4.3 Feature Estimation

LEXenstein's Feature Estimation module allows the calculation of several features for LS related tasks. Its class FeatureEstimator allows the user to select and configure many types of features commonly used by LS approaches.

The FeatureEstimator object can be used either for the creation of LEXenstein's rankers, or in stand-alone setups. For the latter, the class provides a function called *calculateFeatures*, which takes as input a dataset in the VICTOR/CWICTOR format (that can be built from generated/selected substitutions). If the dataset is in VICTOR format, it returns as output a matrix MxN containing M feature values for each of the N substitution candidates listed in the dataset. If the dataset is in CWICTOR format, it returns as output a matrix MxN containing M feature values for the target word of each of the N instances of the dataset. Each of the 11 features supported must be configured individually. They can be grouped in five categories:

Lexicon-oriented: Binary features which receive value 1 if a candidate appears in a given vocabulary, and 0 otherwise.

Morphological: Exploit morphological characteristics of substitutions, such as word length and number of syllables.

Collocational: Estimate n-gram probabilities of form $P\left(S_{h-1}^{h-l}\, c\, S_{h+1}^{h+r}\right)$, where c is a candidate substitution in the hth position in sentence S, and S_{h-1}^{h-l} and S_{h+1}^{h+r} are n-grams of size l and r, respectively.

Sense-oriented: Include several features which relate to the meaning of a candidate substitution, such as: number of senses, lemmas, synonyms, hypernyms, hyponyms and maximum and minimum distance between all of its senses.

Target-dependent: Features that calculate values that represent the relation between a candidate substitution and a target word to be simplified. Includes the similarity between the word vectors that represent the target word and the candidate substitution, and the probability of the target word being translated into the candidate substitution. Note that these features **cannot** be calculated for target words of datasets in CWICTOR format, since it does not include candidate substitutions.

The code snippet below shows an example of how to configure and use a FeatureEstimator object:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.features import *

m = MorphAdornerToolkit('./morph/')
fe = FeatureEstimator()
fe.addLexiconFeature('lexicon.txt', 'Simplicity')
fe.addLengthFeature('Complexity')
fe.addSyllableFeature(m, 'Complexity')
fe.addCollocationalFeature('lm.bin', 2, 2, 'Simplicity')
fe.addSentenceProbabilityFeature('lm.bin', 'Simplicity')
fe.addSenseCountFeature('Simplicity')
fe.addSynonymCountFeature('Simplicity')
fe.addHypernymCountFeature('Simplicity')
fe.addHyponymCountFeature('Simplicity')
fe.addMinDepthFeature('Complexity')
fe.addMaxDepthFeature('Complexity')
fe.addTranslationProbabilityFeature('translation_probs.txt', 'Simplicity')
fe.addWordVectorSimilarityFeature('word_vectors.bin', 'Simplicity')
```

```
feats = fe.calculateFeatures('lexmturk.txt', format='victor')
```

The output produced by the script above will be:

```
[[1.0, 7, 2, 4.470454216003418, 6.206913948059082, 7.3130669593811035,
    10.297701835632324, 12.034161567687988, 13.140314102172852,
    11.422476768493652, 13.158936500549316, 14.26508903503418,
    51.03892517089844, 12, 51, 12, 83, 3, 8, 0.0167457456, 0.234516], [0.0,
    6, 3, 4.487872123718262, 7.389286041259766, 8.941888809204102,
    10.315119743347168, 13.216533660888672, 14.769136428833008,
    11.439894676208496, 14.34130859375, 15.893911361694336, 52.66774368286,
    2, 6, 0, 0, 0, 0, 0.0134387482374, 0.234833]
...
[0.0, 7, 2, 4.950876235961914, 8.496809005737305, 9.830375671386719,
    10.77812385559082, 14.324056625366211, 15.657623291015625,
    11.902898788452148, 15.448831558227539, 16.782398223876953,
    53.556236267089844, 6, 13, 1, 0, 0, 8, 0.0023717232, 0.43812894]]
```

The lexicon file "lexicon.txt" used by a feature in the example above must be a plain text file with one word per line. The language model "lm.bin" used by some features in the example above must be binary, and must be produced by KenLM with the following command lines:

```
lmplz -o [order] <[corpus_of_text] >[language_model_name]
```

```
build_binary [language_model_name] [binary_language_model_name]
```

The translation probabilities file "translation_probs.txt" must be created through the use of fast_align, and estimated over a parallel corpus of sentences. To produce it use the following command line:

```
fast_align -i <parallel_data> -v -d -o <translation_probabilities_file>
```

The fast_align tool can be downloaded from https://github.com/clab/fast_align. The binary word vector model "word_vectors.bin" must be created through the use of Word2Vec. For more information on how to create word vector models, please follow the instructions on the website of the application at https://code.google.com/p/word2vec/.

For more information on how to create a FeatureEstimator object, please refer to LEXenstein's documentation.

4.4 Complex Word Identification

We define Complex Word Identification (CWI) as the task of deciding which words can be considered complex by a certain target audience. There are not many examples in literature of approaches for the task. In [26] the performance of three approaches are compared over the dataset introduced by [27]. The LS strategy of [13] provides an implicit approach for the task.

In the Complex Word Identification module of LEXenstein, one has access to several Complex Word Identification methods. The module contains a series of classes, each representing a distinct method. Currently, it offers support for 5 distinct approaches. The following Sections describe each one individually.

4.4.1 MachineLearningIdentifier

Uses Machine Learning techniques to train classifiers that distinguish between complex and simple words. The class allows for the user to train models with 4 Machine Learning techniques: Support Vector Machines, Decision Trees and linear models estimated with Stochastic Gradient Descent and Passive Aggressive Learning. As input, it requires for a FeatureEstimator object. As output, it produces a vector containing a binary label for the target word of each of the N instances in a dataset in VICTOR or CWICTOR format. The label will have value 1 if the target word was predicted as complex, and 0 otherwise.

Parameters

During instantiation, the MachineLearningIdentifier requires only for a FeatureEstimator object. The user can then use the "calculateTrainingFeatures" and "calculateTestingFeatures" functions to estimate features of training and testing datasets in VICTOR or CWICTOR formats, the "selectKBestFeatures" function to apply feature selection over the features estimated, the "trainSVM", "trainDecisionTreeClassifier", "trainSGDClassifier" and "trainPassiveAggressiveClassifier" functions to train CWI models, and finally the "identifyComplexWords" function to produce output labels. To know more about the parameters and recommended values of the aforementioned functions, please refer to LEXenstein's documentation.

Example

The code snippet below shows the MachineLearningIdentifier class being used:

```
from lexenstein.identifiers import *
```

```
from lexenstein.features import *

fe = FeatureEstimator()
fe.addLexiconFeature('lexicon.txt', 'Simplicity')
fe.addLengthFeature('Complexity')
fe.addSenseCountFeature('Simplicity')

mli = MachineLearningIdentifier(fe)
mli.calculateTrainingFeatures('train_cwictor_corpus.txt')
mli.calculateTestingFeatures('test_victor_corpus.txt')
mli.selectKBestFeatures(k=5)
mli.trainDecisionTreeClassifier()

labels = mli.identifyComplexWords()
```

4.4.2 LexiconIdentifier

Uses a lexicon of complex/simple words in order to judge the complexity of a word: if it appears in the lexicon, then it is complex/simple. As input, it requires for a lexicon file of complex or simple words and expressions. As output, it produces a vector containing a binary label for the target word of each of the N instances in a dataset in VICTOR or CWICTOR format. The label will have value 1 if the lexicon classifies the word as complex, and 0 otherwise.

Parameters

During instantiation, the LexiconIdentifier requires for a lexicon file and an indicator label. The lexicon file must be in plain text and contain one word/expression per line, and the value of the label must be either "complex" or "simple". If the label's value is "complex", then the lexicon will be interpreted as a vocabulary of complex words and expressions, otherwise it will be interpreted as a vocabulary of simple words.

Example

The code snippet below shows the LexiconIdentifier class being used:

```
from lexenstein.identifiers import *
```

```
li = LexiconIdentifier('lexicon.txt', 'simple')

labels = li.identifyComplexWords('test_victor_corpus.txt')
```

4.4.3 ThresholdIdentifier

Estimates the threshold t over a given feature value that best separates complex and simple words. As input, it requires for a FeatureEstimator object. As output, it produces a vector containing a binary label for the target word of each of the N instances in a dataset in VICTOR or CWICTOR format. The label will have value 1 if the lexicon classifies the word as complex, and 0 otherwise.

Parameters

During instantiation, the ThresholdIdentifier requires for a FeatureEstimator object. The user can then use the "calculateTrainingFeatures" and "calculateTestingFeatures" functions to estimate features of training and testing datasets in VICTOR or CWICTOR formats, the "trainIdentifierBruteForce" and "trainIdentifierBinarySearch" functions to train CWI models, and finally the "identifyComplexWords" function to produce output labels.

The "trainIdentifierBruteForce" and "trainIdentifierBinarySearch" functions require for a feature index as input. It will determine over which feature of the ones included in the FeatureEstimator object the threshold t will be estimated. To know more about the parameters and recommended values of the aforementioned functions, please refer to LEXenstein's documentation.

Example

The code snippet below shows the ThresholdIdentifier class being used:

```
from lexenstein.identifiers import *
from lexenstein.features import *

fe = FeatureEstimator()
fe.addLengthFeature('Complexity')
fe.addSenseCountFeature('Simplicity')

ti = ThresholdIdentifier(fe)
ti.calculateTrainingFeatures('train_cwictor_corpus.txt')
```

```
ti.calculateTestingFeatures('test_victor_corpus.txt')
ti.trainIdentifierBruteForce(1)

labels = ti.identifyComplexWords()
```

4.5 Substitution Generation

We define Substitution Generation (SG) as the task of producing candidate substitutions for complex words. Authors commonly address this task by querying WordNet [11] and UMLS[4]. Some examples of authors who resort to this strategy are [10] and [9]. Recently however, learning substitutions from aligned corpora have become a more popular strategy [22] and [13].

In the Substitution Generation module of LEXenstein, one has access to several Substitution Generation methods available in literature. The module contains a series of classes, each representing a distinct approach in literature. Currently, it offers support for 5 distinct approaches. All approaches use LEXenstein's Text Adorning module, described in Section 4.2, to create substitutions for all possible inflections of verbs and nouns. The following Sections describe each one individually.

4.5.1 KauchakGenerator

Employs the strategy described in [13], in which substitutions are automatically extracted from parallel corpora. To be instantiated, this class requires as input an object of the MorphAdornerToolkit class, a parsed document of parallel sentences, the word alignments between them in Pharaoh format, a list of stop words and a NorvigCorrector object. As output, its *getSubstitutions* function produces a dictionary of complex-to-simple substitutions filtered by the criteria described in [13].

Parameters

The parsed parallel document, the alignments file, and the stop words list required by the KauchakGenerator class must be in a specific format. Each line of the parsed parallel document must be in the format described in Example 4.1, where w_i^s is a word in position i

of a source sentence s, p_i^s its POS tag, w_j^t is a word in position j of a target sentence t, and p_j^t its POS tag.

$$\langle w_0^s\rangle|||\langle p_0^s\rangle\cdots\langle w_n^s\rangle|||\langle p_n^s\rangle \quad \langle w_0^t\rangle|||\langle p_0^t\rangle\cdots\langle w_n^t\rangle|||\langle p_n^t\rangle \tag{4.1}$$

All tokens of form $\langle w_i^s\rangle|||\langle p_i^s\rangle$ are separated by a blank space, and the two set of source and target tokens $\langle w_0^s\rangle|||\langle p_0^s\rangle\cdots\langle w_n^s\rangle|||\langle p_n^s\rangle$ are separated by a tabulation marker. An example of file with such notation can be found in "resources/parallel_data/alignment_pos_file.txt".

The alignments file must be in Pharaoh format. Each line of the alignments file must be structured as illustrated in Example 4.2, where $\langle i_h^s\rangle$ is an index i in source sentence s, and $\langle j_h^t\rangle$ is the index j in source sentence t aligned to it.

$$\langle i_0^s\rangle-\langle j_0^t\rangle \ \langle i_1^s\rangle-\langle j_1^t\rangle\cdots\langle i_{n-1}^s\rangle-\langle j_{n-1}^t\rangle \ \langle i_n^s\rangle-\langle j_n^t\rangle \tag{4.2}$$

All tokens of form $\langle i_h^s\rangle-\langle j_h^t\rangle$ are separated by a blank space. An example of file with such notation can be found in "resources/parallel_data/alignments.txt".

Example

The code snippet below shows the KauchakGenerator class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.spelling import *

nc = NorvigCorrector('corpus.txt')

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt',
    nc)
subs = kg.getSubstitutions('lexmturk.txt')
```

4.5.2 YamamotoGenerator

Employs the strategy described in [32], in which substitutions are extracted from dictionary definitions of complex words. This approach requires as input an API key for the Merriam Dictionary[2], which can be obtained for free, and a NorvigCorrector object. As output, it

[2]http://www.dictionaryapi.com/

produces a dictionary linking words in the Merriam Dictionary and WordNet to words with the same Part-of-Speech (POS) tag in its entries' definitions and usage examples.

Parameters

The YamamotoGenerator class requires a free Dictionary key to the Merriam Dictionary. To get the key, follow the steps below:

1. Visit the page http://www.dictionaryapi.com/register/index.htm.
2. Fill in your personal information.
3. In "Request API Key #1:" and "Request API Key #2:", select "Collegiate Dictionary" and "Collegiate Thesaurus".
4. Login in http://www.dictionaryapi.com.
5. Visit your "My Keys" page.
6. Use the "Dictionary" key to create a YamamotoGenerator object.

Example

The code snippet below shows the YamamotoGenerator class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.spelling import *

nc = NorvigCorrector('corpus.txt')

m = MorphAdornerToolkit('./morph/')

yg = YamamotoGenerator(m, '0000-0000-0000-0000', nc)
subs = yg.getSubstitutions('lexmturk.txt')
```

4.5.3 MerriamGenerator

Extracts a dictionary linking words to their synonyms, as listed in the Merriam Thesaurus. This approach requires as input an API key for the Merriam Thesaurus[3], which can be obtained for free, and a NorvigCorrector object.

Parameters

The MerriamGenerator class requires a free Thesaurus key to the Merriam Dictionary. To get the key, follow the steps below:

1. Visit the page http://www.dictionaryapi.com/register/index.htm.
2. Fill in your personal information.
3. In "Request API Key #1:" and "Request API Key #2:", select "Collegiate Dictionary" and "Collegiate Thesaurus".
4. Login in http://www.dictionaryapi.com.
5. Visit your "My Keys" page.
6. Use the "Thesaurus" key to create a MerriamGenerator object.

Example

The code snippet below shows the MerriamGenerator class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.spelling import *

nc = NorvigCorrector('corpus.txt')

m = MorphAdornerToolkit('./morph/')

mg = MerriamGenerator(m, '0000-0000-0000-0000', nc)
subs = mg.getSubstitutions('lexmturk.txt')
```

[3] http://www.dictionaryapi.com/

4.5.4 WordnetGenerator

Extracts a dictionary linking words to their synonyms, as listed in WordNet. It requires for a NorvigCorrector object, the path to a POS tagging model, and the path to the Stanford Tagger [17].

Parameters

In order to obtain the model and tagger required by the WordnetGenerator class, download the full version of the Stanford Tagger package from the link: http://nlp.stanford.edu/software/tagger.shtml. Inside the package's "models" folder you will find tagging models for various languages. In the package's root folder, you will find the "stanford-postagger.jar" application, which is the one required by the WordnetGenerator.

Example

The code snippet below shows the WordnetGenerator class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.spelling import *

nc = NorvigCorrector('corpus.txt')

m = MorphAdornerToolkit('./morph/')

wg = WordnetGenerator(m, nc, 'english-left3words-distsim.tagger',
    'stanford-postagger.jar')
subs = wg.getSubstitutions('lexmturk.txt')
```

4.5.5 BiranGenerator

Employs the strategy described in [2], in which substitutions are filtered from the Cartesian product between vocabularies of complex and simple words. This approach requires as input vocabularies of complex and simple words, as well as two Language Models trained over complex and simple corpora, a NorvigCorrector object, the path to a POS tagging model, and the path to the Stanford Tagger [17]. As output, it produces a dictionary linking words to a set of synonyms and hypernyms filtered by the criteria described in [2].

Parameters

The vocabularies of complex and simple words and Language Models trained over complex and simple corpora required by the BiranGenerator class must be in a specific format. The vocabularies must contain one word per line, and can be produced over large corpora with SRILM by running the following command line:

```
ngram-count -text [corpus_of_text] -write-vocab [vocabulary_name]
```

The Language Models must be binary, and must be produced by KenLM with the following command lines:

```
lmplz -o [order] <[corpus_of_text] >[language_model_name]
```

```
build_binary [language_model_name] [binary_language_model_name]
```

Complex and simple data can be downloaded fro David Kauchak's page[4], or extracted from other sources. In order to obtain the model and tagger required by the BiranGenerator class, download the full version of the Stanford Tagger package from the link: http://nlp.stanford.edu/software/tagger.shtml. Inside the package's "models" folder you will find tagging models for various languages. In the package's root folder, you will find the "stanford-postagger.jar" application, which is the one required by the BiranGenerator.

Example

The code snippet below shows the BiranGenerator class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.spelling import *

nc = NorvigCorrector('corpus.txt')

m = MorphAdornerToolkit('./morph/')

bg = BiranGenerator(m, 'vocabc.txt', 'vocabs.txt', 'lmc.bin', 'lms.bin', nc,
    'english-left3words-distsim.tagger', 'stanford-postagger.jar')
subs = bg.getSubstitutions('lexmturk.txt')
```

[4]http://www.cs.pomona.edu/ dkauchak/simplification/

4.6 Substitution Selection

Substitution Selection (SS) is the task of selecting which substitutions from a given list can replace a complex word in a given sentence without altering its meaning. Most work addresses this task referring to the context of the complex word by employing Word Sense Disambiguation (WSD) approaches [20, 25], or by discarding substitutions which do not share the same POS tag of the target complex word [22, 32].

LEXenstein's SS module provides access to 8 approaches, each represented by a Python class. All classes have a "selectCandidates" function, which receives as input a set of candidate substitutions and a corpus in the VICTOR format. The candidate substitutions can be either a dictionary produced by a Substitution Generation approach, or a list of candidates that have been already selected by another Substitution Selection approach. This feature allows for multiple selectors to be used in sequence. The following Sections describe each of the classes in the LEXenstein SS module individually.

4.6.1 WordVectorSelector

Employs a novel strategy, in which a word vector model is used to determine which substitutions have the closest meaning to that of the sentence being simplified. It retrieves a user-defined percentage of the substitutions, which are ranked with respect to the cosine distance between its word vector and the sum of some, or all of the sentences' words, depending on the settings defined by the user.

Parameters

To create a WordVectorSelector object, you must provide a binary word vector model created with Word2Vec. To create the word vector, follow the steps below:

1. Download and install Word2Vec in your machine from https://code.google.com/p/word2vec/.

2. Gather large amounts of corpora (>10 billion words). You can find some sources of data in https://code.google.com/p/word2vec/.

3. With Word2Vec installed, run it with the following command line:

```
./word2vec -train <corpus> -output <binary_model_path> -cbow 1 -size
    300 -window 5 -negative 3 -hs 0 -sample 1e-5 -threads 12 -binary 1
    -min-count 10
```

The command line above creates word vectors with 300 dimensions and considers only a window of 5 tokens around each word. You can customize those parameters if you wish. For more information on how to use other class functions, please refer to LEXenstein's documentation.

Example

The code snippet below shows the WordVectorSelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.selectors import *

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

wordvecselector = WordVectorSelector('word_vector_model.bin')
selected = wordvecselector.selectCandidates(subs, 'lexmturk.txt',
    proportion=0.75, stop_words_file='stop_words.txt', onlyInformative=True,
    keepTarget=True, onePerWord=True)
```

4.6.2 BiranSelector

Employs the strategy described in [2], in which a word co-occurrence model is used to determine which substitutions have meaning similar to that of a target complex word. It filters all substitutions which are estimated to be more complex than the target word, and also all those for which the distance between its co-occurrence vector and the target sentence's vector is higher than a threshold set by the user.

Parameters

To create a BiranSelector object, you must provide a word co-occurrence model. The model must be in plain text format, and each line must follow the format illustrated in Example 4.5, where $\langle w_i \rangle$ is a word, $\left\langle c_i^j \right\rangle$ a co-occurring word and $\left\langle f_i^j \right\rangle$ its frequency of appearance.

$$\langle w_i \rangle \ \left\langle c_i^0 \right\rangle : \left\langle f_i^0 \right\rangle \ \left\langle c_i^1 \right\rangle : \left\langle f_i^1 \right\rangle \cdots \left\langle c_i^{n-1} \right\rangle : \left\langle f_i^{n-1} \right\rangle \ \left\langle c_i^n \right\rangle : \left\langle f_i^n \right\rangle \tag{4.3}$$

Each component in the format above must be separated by a tabulation marker. To create a co-occurrence model, either create a script that does so, or follow the steps below:

1. Gather a corpus of text composed of one tokenized and truecased sentence per line.
2. Run the script resources/scripts/Produce_Co-occurrence_Model.py with the following command line:

```
python Produce_Co-occurrence_Model.py <corpus> <window> <model_path>
```

   Where "<window>" is the number of tokens to the left and right of a word to be included as a co-occurring word.

To produce models faster, you can split your corpus in various small portions, run parallel processes to produce various small models, and then join them. For more information on how to use other class functions, please refer to LEXenstein's documentation.

Example

The code snippet below shows the BiranSelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.selectors import *

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

biranselector = BiranSelector('cooc_model.txt')
selected = biranselector.selectCandidates(subs, 'lexmturk.txt', 0.01, 0.75)
```

4.6.3 WSDSelector

Allows for the user to use many distinct classic WSD approaches in SS. It requires for the PyWSD [30] module to be installed, which includes the approaches presented by [18] and [31], as well as baselines such as random and first senses. The user can use any of the aforementioned approaches through the WSDSelector class by changing instantiation parameters.

Parameters

During instantiation, the WSDSelector class requires only for you to provide an id for the WSD method that you desire to use for Substitution Selection. For the options available, please refer to LEXenstein's documentation.

Example

The code snippet below shows the WSDSelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.selectors import *

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

wsdselector = WSDSelector('lesk')
selected = wsdselector.selectCandidates(subs, 'lexmturk.txt')
```

4.6.4 POSTagSelector

Selects only those candidates which share the same POS tag of a given target word in a sentence. The selector initially parses a given sentence, and retrieves the POS tag of the target word. It then replaces the target word in the sentence with each candidate substitution, parses the resulting sentence, and filters any candidates which have a POS tag that is not equal to the one of the target word.

Parameters

During instantiation, the POSTagSelector class requires no parameters.

Example

The code snippet below shows the POSTagSelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
```

```
from lexenstein.selectors import *

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

posselector = POSTagSelector()
selected = posselector.selectCandidates(subs, 'lexmturk.txt')
```

4.6.5 ClusterSelector

Selects only those candidates which appear in the same word clusters in which a given target word is present. This strategy is inspired by the work of [1], in which synonyms are automatically extracted from a latent variable language model.

Parameters

During instantiation, the ClusterSelector class requires for a file with clusters of words. The file must be in plain text format, and each line must follow the format illustrated in Example 4.4, which is the one adopted by Brown Clusters [6] implementation of [19]. In the Example 4.4, c_i is a class identifier, w_i a word, and f_i an optional frequency of occurrence of word w_i in the corpus over which the word classes were estimated.

$$\langle c_i \rangle \ \langle w_i \rangle \langle f_i \rangle \tag{4.4}$$

Each component in the format above must be separated by a tabulation marker. To create the file, one can use the Brown Clusters implementation of [19]. Once the tool is installed, run the following command line:

```
./wcluster --text <corpus_of_sentences> --c <number_of_clusters>
```

The clusters file will be placed at **input-c50-p1.out/paths**.

Example

The code snippet below shows the ClusterSelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
```

```
from lexenstein.generators import *
from lexenstein.selectors import *

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

clusterselector = ClusterSelector('clusters.txt')
selected = clusterselector.selectCandidates(subs, 'lexmturk.txt')
```

4.6.6 BoundarySelector

Employs a novel strategy, in which a Boundary Ranker is trained over a given set of features and then used to rank candidate substitutions according to how likely they are of being able to replace a target word without compromising the sentence's grammaticality or coherence. It retrieves a user-defined percentage of a set of substitutions.

Parameters

During instantiation, the BoundarySelector class requires for a BoundaryRanker object, which must be configured according to which features and resources the user intends to use to rank substitution candidates. The user can then use the "trainSelector" function to train the selector given a set of parameters, or the "trainSelectorWithCrossValidation" function to train it with cross-validation. Finally, the user can then retrieve a proportion of the candidate substitutions by using the "selectCandidates" function. For more information about the parameters of each function, please refer to LEXenstein's documentation.

Example

The code snippet below shows the BoundarySelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.selectors import *
from lexenstein.features import *
from lexenstein.rankers import *
```

```
fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')
fe.addSenseCountFeature('Simplicity')

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

br = BoundaryRanker(fe)
bs = BoundarySelector(br)
bs.trainSelectorWithCrossValidation('lexmturk.txt', 1, 5, 0.25)
selected = bs.selectCandidates(subs, 'lexmturk.txt', 'temp.txt', 0.25)
```

4.6.7 SVMRankSelector

Employs a novel strategy, in which a SVM Ranker is trained over a given set of features and then used to rank candidate substitutions according to how likely they are of being able to replace a target word without compromising the sentence's grammaticality or coherence. It retrieves a user-defined percentage of a set of substitutions.

Parameters

During instantiation, the SVMRankSelector class requires for a SVMRanker object, which must be configured according to which features and resources the user intends to use to rank substitution candidates. The user can then use the "trainSelector" function to train the selector given a set of parameters, or the "trainSelectorWithCrossValidation" function to train it with cross-validation. Finally, the user can then retrieve a proportion of the candidate substitutions by using the "selectCandidates" function. For more information about the parameters of each function, please refer to LEXenstein's documentation.

Example

The code snippet below shows the SVMRankSelector class being used:

```
from lexenstein.morphadorner import MorphAdornerToolkit
from lexenstein.generators import *
from lexenstein.selectors import *
```

```
from lexenstein.features import *
from lexenstein.rankers import *

fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')
fe.addSenseCountFeature('Simplicity')

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')

sr = SVMRanker(fe, './svm-rank/'
ss = SVMRankSelector(br)
ss.trainSelectorWithCrossValidation('lexmturk.txt', 'f1.txt', 'm.txt', 5,
    0.25, './temp/', 0)
selected = bs.selectCandidates(subs, 'lexmturk.txt', 'f2.txt', 's1.txt',
    'temp.txt', 0.25)
```

4.6.8 VoidSelector

Does not perform any type of explicit substitution selection, and selects all possible substitutions generated for a given target word.

Parameters

During instantiation, the VoidSelector class requires no parameters.

Example

The code snippet below shows the VoidSelector class being used:

```
from lexenstein.generators import *
from lexenstein.selectors import *

m = MorphAdornerToolkit('./morph/')

kg = KauchakGenerator(m, 'parallel.txt', 'alignments.txt', 'stop_words.txt')
subs = kg.getSubstitutions('lexmturk.txt')
```

```
voidselector = VoidSelector()
selected = voidselector.selectCandidates(subs, 'lexmturk.txt')
```

4.7 Substitution Ranking

Substitution Ranking (SR) is the task of ranking a set of selected substitutions for a target complex word with respect to its simplicity. Approaches vary from simple word length and frequency-based measures [2, 8–10] to more sophisticated linear combination as scoring functions [14] and Machine Learning approaches [13].

LEXenstein's SR module provides access to 6 approaches, each represented by a Python class. The following Sections describe each one individually.

4.7.1 MetricRanker

Employs a simple ranking strategy based on the values of a single feature provided by the user. By configuring the input FeatureEstimator object, the user can calculate values of several features for the candidates in a given dataset, and easily rank the candidates according to each of these features.

Parameters

During instantiation, the MetricRanker requires only a configured FeatureEstimator object, which must contain at least one feature that can be used as a metric for ranking. Once created, the user can retrieve rankings by using MetricRanker's "getRankings" function, which requires a VICTOR corpus and the index of a feature to be used as a ranking metric.

Example

The code snippet below shows the WSDSelector class being used:

```
from lexenstein.features import *
from lexenstein.rankers import *

fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')
fe.addLengthFeature('Complexity')
fe.addSenseCountFeature('Simplicity')
```

```
mr = MetricRanker(fe)
frequency_ranks = mr.getRankings('lexmturk.txt', 0)
length_ranks = mr.getRankings('lexmturk.txt', 1)
sense_ranks = mr.getRankings('lexmturk.txt', 2)
```

4.7.2 SVMRanker

Use Support Vector Machines in a setup that minimizes a loss function with respect to a ranking model. In LS, this strategy is the one employed in the experiments of [13], yielding promising results.

Parameters

During instantiation, the SVMRanker requires for a FeatureEstimator object and a path to the root installation folder of SVM-Rank [15]. The user can then use the "getFeaturesFile", "getTrainingModel", "getScoresFile" and "getRankings" to train SVM ranking models and rank candidate substitutions. For more information on these functions' parameters, please refer to LEXenstein's documentation and the example in the following Section.

Example

The code snippet below shows the SVMRanker class being used:

```
from lexenstein.features import *
from lexenstein.rankers import *

fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')
fe.addLengthFeature('Complexity')
fe.addSenseCountFeature('Simplicity')

svmr = SVMRanker(fe, '/svm-rank/')
svmr.getFeaturesFile('lexmturk.txt', 'features.txt')
svmr.getTrainingModel('features.txt', 0.1, 0.1, 0, 'model.txt')
svmr.getScoresFile('features.txt', 'model.txt', 'scores.txt')
rankings = svmr.getRankings('features.txt', 'scores.txt')
```

4.7.3 BoundaryRanker

Employs a novel strategy, in which ranking is framed as a binary classification task. During training, this approach assigns the label 1 to all candidates of rank $1 \geq r \geq p$, where p is a range set by the user, and 0 to the remaining candidates. It then trains a stochastic descent linear classifier based on the features specified in the FeatureEstimator object. During testing, candidate substitutions are ranked based on how far from 0 they are.

Parameters

During instantiation, the BoundaryRanker requires only for a FeatureEstimator object. The user can then use the "trainRanker" function to train a ranking model, and the "getRankings" to rank the candidates of a VICTOR corpus. For more information on the training parameters supported by the "trainRanker" function, please refer to LEXenstein's documentation.

Example

The code snippet below shows the BoundaryRanker class being used:

```
from lexenstein.features import *
from lexenstein.rankers import *

fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')
fe.addLengthFeature('Complexity')
fe.addSenseCountFeature('Simplicity')

br = BoundaryRanker(fe)
br.trainRanker('lexmturk.txt', 1, 'modified_huber', 'l1', 0.1, 0.1, 0.001)
rankings = br.getRankings('lexmturk.txt')
```

4.7.4 BiranRanker

Employs the strategy of [2], which models complexity as a function of a word's length and frequency in corpora of complex and simple content. As input, it requires for a language model trained over complex data, and a language model trained over simple data.

Parameters

The language models required by the BiranRanker must be binary, and must be produced by KenLM with the following command lines:

```
lmplz -o [order] <[corpus_of_text] >[language_model_name]
```

```
build_binary [language_model_name] [binary_language_model_name]
```

Complex and simple data can be downloaded fro David Kauchak's page[5], or extracted from other sources.

Example

The code snippet below shows the BiranRanker class being used:

```
from lexenstein.features import *
from lexenstein.rankers import *

br = BiranRanker('lmc.bin', 'lms.bin')
rankings = br.getRankings('lexmturk.txt')
```

4.7.5 YamamotoRanker

Employs the strategy of [32], which ranks words according to a weighted function that considers their frequency in a corpus of simple content, co-occurrence frequency with a target complex word, sense distance, point-wise mutual information and trigram frequencies. As input, it requires for a language model trained over a corpus of simple data and a co-occurrence model.

Parameters

The language model required by the YamamotoRanker must be binary, and must be produced by KenLM with the following command lines:

```
lmplz -o [order] <[corpus_of_text] >[language_model_name]
```

```
build_binary [language_model_name] [binary_language_model_name]
```

[5]http://www.cs.pomona.edu/ dkauchak/simplification/

Complex and simple data can be downloaded fro David Kauchak's page[6], or extracted from other sources. The co-occurrence model must be in plain text format, and each line must follow the format illustrated in Example 4.5, where $\langle w_i \rangle$ is a word, $\left\langle c_i^j \right\rangle$ a co-occurring word and $\left\langle f_i^j \right\rangle$ its frequency of appearance.

$$\langle w_i \rangle \ \langle c_i^0 \rangle : \langle f_i^0 \rangle \ \langle c_i^1 \rangle : \langle f_i^1 \rangle \cdots \langle c_i^{n-1} \rangle : \langle f_i^{n-1} \rangle \ \langle c_i^n \rangle : \langle f_i^n \rangle \tag{4.5}$$

Each component in the format above must be separated by a tabulation marker. To create a co-occurrence model, either create a script that does so, or follow the steps below:

1. Gather a corpus of text composed of one tokenized and truecased sentence per line.

2. Run the script resources/scripts/Produce_Co-occurrence_Model.py with the following command line:

   ```
   python Produce_Co-occurrence_Model.py <corpus> <window> <model_path>
   ```

   Where "<window>" is the number of tokens to the left and right of a word to be included as a co-occurring word.

To produce models faster, you can split your corpus in various small portions, run parallel processes to produce various small models, and then join them. For more information on the parameters of each function of the YamamotoRanker class, please refer to the LEXenstein documentation.

Example

The code snippet below shows the YamamotoRanker class being used:

```
from lexenstein.features import *
from lexenstein.rankers import *

yr = YamamotoRanker('lms.bin', 'cooc_model.txt')
rankings = yr.getRankings('lexmturk.txt')
```

[6] http://www.cs.pomona.edu/ dkauchak/simplification/

4.7.6 BottRanker

Employs the strategy of [5], which ranks words according to a weighted function that considers their length and frequency in a corpus of simple content. As input, it requires for a language model trained over a corpus of simple data.

Parameters

The language model required by the BottRanker must be binary, and must be produced by KenLM with the following command lines:

```
lmplz -o [order] <[corpus_of_text] >[language_model_name]
```

```
build_binary [language_model_name] [binary_language_model_name]
```

Complex and simple data can be downloaded fro David Kauchak's page[7], or extracted from other sources.

Example

The code snippet below shows the BottRanker class being used:

```
from lexenstein.features import *
from lexenstein.rankers import *

br = BottRanker('lms.bin')
rankings = br.getRankings('lexmturk.txt')
```

4.8 Evaluation

Since one of the goals of LEXenstein is to facilitate the benchmarking LS approaches, it is crucial that it provides evaluation methods. This module includes functions for the evaluation of all sub-tasks, both individually and in combination. It contains 5 classes, each designed for one form of evaluation. We discuss them in more detail in the following Sections.

[7] http://www.cs.pomona.edu/ dkauchak/simplification/

4.8.1 IdentifierEvaluator

Provides evaluation metrics for CWI methods. It requires a gold-standard in the CWICTOR format and a set of binary word complexity labels. The labels must have value 1 for complex words, and 0 otherwise. It returns the Precision, Recall and F-measure, where Precision is the proportion of correctly labeled words, Recall the proportion of complex words identified, and F-measure their harmonic mean.

The code snippet below shows the IdentifierEvaluator class being used:

```
from lexenstein.identifiers import *
from lexenstein.evaluators import *

li = LexiconIdentifier('lexicon.txt', 'simple')
labels = li.identifyComplexWords('cwictor_gold.txt')

ie = IdentifierEvaluator()
precision, recall, fmeasure = ie.evaluateIdentifier('cwictor_gold.txt',
    labels)
```

4.8.2 GeneratorEvaluator

Provides evaluation metrics for SG methods. It requires a gold-standard in the VICTOR format and a set of generated substitutions. It returns the Potential, Precision, Recall and F-measure, where Potential is the proportion of instances in which at least one of the substitutions generated is present in the gold-standard, Precision the proportion of generated instances which are present in the gold-standard, Recall the proportion of gold-standard candidates that were generated, and F-measure the harmonic mean between Precision and Recall.

The code snippet below shows the GeneratorEvaluator class being used:

```
from lexenstein.generators import *
from lexenstein.evaluators import *
from lexenstein.morphadorner import *

m = MorphAdornerToolkit('./morph/')

kg = WordnetGenerator(m)
subs = kg.getSubstitutions('lexmturk.txt')
```

```
ge = GeneratorEvaluator()
potential, precision, recall, fmeasure =
    ge.evaluateGenerator('lexmturk.txt', subs)
```

4.8.3 SelectorEvaluator

Provides evaluation metrics for SS methods. It requires a gold-standard in the VICTOR format and a set of selected substitutions. It returns the Potential, Precision and F-measure of the SS approach, where Potential is the proportion of instances in which at least one of the substitutions selected is present in the gold-standard, Precision the proportion of selected candidates which are present in the gold-standard, Recall the proportion of gold-standard candidates that were selected, and F-measure the harmonic mean between Precision and Recall.

The code snippet below shows the SelectorEvaluator class being used:

```
from lexenstein.generators import *
from lexenstein.selectors import *
from lexenstein.evaluators import *
from lexenstein.morphadorner import *

m = MorphAdornerToolkit('./morph/')

kg = WordnetGenerator(m)
subs = kg.getSubstitutions('lexmturk.txt')

biranselector = BiranSelector('cooc_model.txt')
selected = biranselector.selectCandidates(subs, 'lexmturk.txt', 0.01, 0.75)

se = SelectorEvaluator()
potential, precision, recall, fmeasure = se.evaluateSelector('lexmturk.txt',
    selected)
```

4.8.4 RankerEvaluator

Provides evaluation metrics for SR methods. It requires a gold-standard in the VICTOR format and a set of ranked substitutions. It returns the TRank-at-1 : 3 and Recall-at-1 : 3 metrics [29], where Trank-at-i is the proportion of instances in which a candidate of gold-rank

$r \leq i$ was ranked first, and Recall-at-i the proportion of candidates of gold-rank $r \leq i$ that are ranked in positions $p \leq i$.

The code snippet below shows the RankerEvaluator class being used:

```
from lexenstein.rankers import *
from lexenstein.features import *

fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')

mr = MetricRanker(fe)
rankings = mr.getRankings('lexmturk.txt', 0)

re = RankerEvaluator()
t1, t2, t3, r1, r2, r3 = re.evaluateRanker('lexmturk.txt', rankings)
```

4.8.5 PipelineEvaluator

Provides evaluation metrics for the entire LS pipeline. It requires as input a gold-standard in VICTOR format and a set of ranked substitutions which have been generated and selected by a given set of approaches. It returns the approaches' Precision, Accuracy and Change Proportion, where Precision is the proportion of instances in which the highest ranking substitution is not the target complex word itself and is in the gold-standard, Accuracy is the proportion of instances in which the highest ranking substitution is in the gold-standard, and Change Proportion is the proportion of instances in which the highest ranking substitution is not the target complex word itself.

The code snippet below shows the PipelineEvaluator class being used:

```
from lexenstein.generators import *
from lexenstein.selectors import *
from lexenstein.evaluators import *
from lexenstein.rankers import *
from lexenstein.features import *
from lexenstein.morphadorner import *

m = MorphAdornerToolkit('./morph/')

kg = WordnetGenerator(m)
```

```
subs = kg.getSubstitutions('lexmturk.txt')

bs = BiranSelector('cooc_model.txt')
selected = bs.selectCandidates(subs, 'lexmturk.txt', 0.01, 0.75)
bs.toVictorFormat('lexmturk.txt', selected, 'victor.txt',
    addTargetAsCandidate=True)

fe = FeatureEstimator()
fe.addCollocationalFeature('lm.txt', 0, 0, 'Complexity')

mr = MetricRanker(fe)
rankings = mr.getRankings('victor.txt', 0)

pe = PipelineEvaluator()
precision, accuracy, changed = pe.evaluatePipeline('lexmturk.txt', rankings)
```

References

[1] Belder, J. D. and Moens, M. (2010). Text simplification for children. *Prroceedings of the SIGIR workshop on . . . .*

[2] Biran, O., Brody, S., and Elhadad, N. (2011). Putting it Simply: a Context-Aware Approach to Lexical Simplification. *The 49th Annual Meeting of the ACL.*

[3] Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly.

[4] Bodenreider, O. (2004). The unified medical language system (umls): integrating biomedical terminology. *Nucleic acids research*.

[5] Bott, S., Rello, L., Drndarevic, B., and Saggion, H. (2012). Can Spanish Be Simpler ? LexSiS : Lexical Simplification for Spanish ¿ Puede ser el Español más simple ? LexSiS : Simplificación Léxica en Español.

[6] Brown, P. F., deSouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational Linguistics*, 18:467–479.

[7] Burns, P. R. (2013). MorphAdorner v2: A Java Library for the Morphological Adornment of English Language Texts.

[8] Carroll, J., Minnen, G., Canning, Y., Devlin, S., and Tait, J. (1998). Practical simplification of english newspaper text to assist aphasic readers. In *Proceedings of AAAI-98*.

[9] Carroll, J., Minnen, G., Pearce, D., Canning, Y., Devlin, S., and Tait, J. (1999). Simplifying Text for Language Impaired Readers. *The 9th Conference of EACL.*

[10] Devlin, S. and Tait, J. (1998). The use of a psycholinguistic database in the simplification of text for aphasic readers. *Linguistic Databases*.

[11] Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Bradford Books.

[12] Heafield, K., Pouzyrevsky, I., Clark, J. H., and Koehn, P. (2013). Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria.

[13] Horn, C., Manduca, C., and Kauchak, D. (2014). Learning a Lexical Simplifier Using Wikipedia. *The 52nd Annual Meeting of the ACL.*

[14] Jauhar, S. and Specia, L. (2012). UOW-SHEF: SimpLex–lexical simplicity ranking based on contextual and psycholinguistic features. *Proceedings of the 1st *SEM*.

[15] Joachims, T. (2002). Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM*.

[16] Jones, E., Oliphant, T., Peterson, P., et al. (2015). SciPy: Open source scientific tools for Python.

[17] Klein, D. and Manning, C. D. (2003). Accurate Unlexicalized Parsing. In *Proceedings of the 41ST Annual Meeting of ACL*.

[18] Lesk, M. (1986). Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of the 5th Annual SIGDOC*.

[19] Liang, P. (2012). C++ implementation of the brown word clustering algorithm. https://github.com/percyliang/brown-cluster.

[20] Nunes, B. P., Kawase, R., Siehndel, P., Casanova, M. a., and Dietze, S. (2013). As Simple as It Gets - A Sentence Simplifier for Different Learning Levels and Contexts. *IEEE 13th ICALT*.

[21] Paetzold, G. H. (2015). Morph adorner toolkit: Morph adorner made simple. http://ghpaetzold.github.io/MorphAdornerToolkit/.

[22] Paetzold, G. H. and Specia, L. (2013). Text simplification as tree transduction. In *Proceedings of the 9th STIL*.

[23] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[24] Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA.

[25] Sedding, J. and Kazakov, D. (2004). Wordnet-based text document clustering. In *Proceedings of the 3rd ROMAND*.

[26] Shardlow, M. (2013a). A Comparison of Techniques to Automatically Identify Complex Words. *ACL (Student Research Workshop)*.

[27] Shardlow, M. (2013b). *Proceedings of the Second Workshop on Predicting and Improving Text Readability for Target Reader Populations*, chapter The CW Corpus: A New Resource for Evaluating the Identification of Complex Words, pages 69–77. Association for Computational Linguistics.

[28] Shelley, M. (2007). *Frankenstein*. Pearson Education.

[29] Specia, L., Jauhar, S. K., and Mihalcea, R. (2012). Semeval-2012 task 1: English lexical simplification. In *Proceedings of the 1st *SEM*.

[30] Tan, L. (2014). Pywsd: Python implementations of word sense disambiguation (wsd) technologies [software]. https://github.com/alvations/pywsd.

[31] Wu, Z. and Palmer, M. (1994). Verbs semantics and lexical selection. In *Proceedings of the 32nd Annual Meeting of ACL.*

[32] Yamamoto, T. (2013). Selecting Proper Lexical Paraphrase for Children.