pyspellchecker Documentation

Release 0.7.2

Tyler Barrus

Sep 22, 2023

Contents

Pure Python Spell Checking based on Peter Norvig's blog post on setting up a simple spell checking algorithm.

It uses a Levenshtein Distance algorithm to find permutations within an edit distance of 2 from the original word. It then compares all permutations (insertions, deletions, replacements, and transpositions) to known words in a word frequency list. Those words that are found more often in the frequency list are **more likely** the correct results.

`pyspellchecker` supports multiple languages including English, Spanish, German, French, Portuguese, Arabic and Basque. For information on how the dictionaries were created and how they can be updated and improved, please see the **Dictionary Creation and Updating** section of the readme!

`pyspellchecker` supports **Python 3**

`pyspellchecker` allows for the setting of the Levenshtein Distance (up to two) to check. For longer words, it is highly recommended to use a distance of 1 and not the default 2. See the quickstart to find how one can change the distance parameter.

CHAPTER 1

Installation

The easiest method to install is using pip:

```
pip install pyspellchecker
```

To build from source:

```
git clone https://github.com/barrust/pyspellchecker.git
cd pyspellchecker
python -m build
```

For *python 2.7* support, install release 0.5.6 but note that no future updates will support *python 2*.

```
pip install pyspellchecker==0.5.6
```

CHAPTER 2

Quickstart

After installation, using `pyspellchecker` should be fairly straight forward:

```
from spellchecker import SpellChecker

spell = SpellChecker()

# find those words that may be misspelled
misspelled = spell.unknown(['something', 'is', 'hapenning', 'here'])

for word in misspelled:
    # Get the one `most likely` answer
    print(spell.correction(word))

    # Get a list of `likely` options
    print(spell.candidates(word))
```

If the Word Frequency list is not to your liking, you can add additional text to generate a more appropriate list for your use case.

```
from spellchecker import SpellChecker

spell = SpellChecker()  # loads default word frequency list
spell.word_frequency.load_text_file('./my_free_text_doc.txt')

# if I just want to make sure some words are not flagged as misspelled
spell.word_frequency.load_words(['microsoft', 'apple', 'google'])
spell.known(['microsoft', 'google'])  # will return both now!
```

If the words that you wish to check are long, it is recommended to reduce the *distance* to 1. This can be accomplished either when initializing the spell check class or after the fact.

```
from spellchecker import SpellChecker

spell = SpellChecker(distance=1)  # set at initialization
```

(continues on next page)

(continued from previous page)

```
# do some work on longer words

spell.distance = 2  # set the distance parameter back to the default
```

CHAPTER 3

Non-English Dictionaries

`pyspellchecker` supports several default dictionaries as part of the default package. Each is simple to use when initializing the dictionary:

```
from spellchecker import SpellChecker

english = SpellChecker()  # the default is English (language='en')
spanish = SpellChecker(language='es')  # use the Spanish Dictionary
russian = SpellChecker(language='ru')  # use the Russian Dictionary
arabic = SpellChecker(language='ar')   # use the Arabic Dictionary
```

The currently supported dictionaries are:

- English - 'en'
- Spanish - 'es'
- French - 'fr'
- Portuguese - 'pt'
- German - 'de'
- Russian - 'ru'
- Arabic - 'ar'
- Basque - 'eu'
- Latvian - 'lv'

CHAPTER 4

Dictionary Creation and Updating

The creation of the dictionaries is, unfortunately, not an exact science. I have provided a script that, given a text file of sentences (in this case from OpenSubtitles) it will generate a word frequency list based on the words found within the text. The script then attempts to ***clean up*** the word frequency by, for example, removing words with invalid characters (usually from other languages), removing low count terms (misspellings?) and attempts to enforce rules as available (no more than one accent per word in Spanish). Then it removes words from a list of known words that are to be removed. It then adds words into the dictionary that are known to be missing or were removed for being too low frequency.

The script can be found here: `scripts/build_dictionary.py`. The original word frequency list parsed from OpenSubtitles can be found in the `scripts/data/` folder along with each language's *include* and *exclude* text files.

Any help in updating and maintaining the dictionaries would be greatly desired. To do this, a discussion could be started on GitHub or pull requests to update the include and exclude files could be added.

CHAPTER 5

Additional Methods

On-line documentation is available; below contains the cliff-notes version of some of the available functions:

`correction(word)`: Returns the most probable result for the misspelled word

`candidates(word)`: Returns a set of possible candidates for the misspelled word

`known([words])`: Returns those words that are in the word frequency list

`unknown([words])`: Returns those words that are not in the frequency list

`word_probability(word)`: The frequency of the given word out of all words in the frequency list

5.1 The following are less likely to be needed by the user but are available:

`edit_distance_1(word)`: Returns a set of all strings at a Levenshtein Distance of one based on the alphabet of the selected language

`edit_distance_2(word)`: Returns a set of all strings at a Levenshtein Distance of two based on the alphabet of the selected language

CHAPTER 6

Credits

- Peter Norvig blog post on setting up a simple spell checking algorithm
- P Lison and J Tiedemann, 2016, OpenSubtitles2016: Extracting Large Parallel Corpora from Movie and TV Subtitles. In Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)

CHAPTER 7

Table of contents

7.1 Quickstart

`pyspellchecker` is designed to be easy to use to get basic spell checking.

7.1.1 Installation

The best experience is likely to use `pip`:

```
pip install pyspellchecker
```

If you are using virtual environments, it is recommended to use `pipenv` to combine pip and virtual environments:

```
pipenv install pyspellchecker
```

Read more about Pipenv

7.1.2 Basic Usage

Setting up the spell checker requires importing and initializing the instance.

```
from spellchecker import SpellChecker

spell = SpellChecker()
```

There are several methods to determine if a word is in the word frequency list:

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell['morning']  # True
```

(continues on next page)

(continued from previous page)

```
'morning' in spell  # True

# find those words from a list of words that are found in the dictionary
spell.known(['morning', 'hapenning'])  # {'morning'}

# find those words from a list of words that are not found in the dictionary
spell.unknown(['morning', 'hapenning'])  # {'hapenning'}
```

Once a word is identified as misspelled, you can find the likeliest replacement:

```
from spellchecker import SpellChecker

spell = SpellChecker()

misspelled = spell.unknown(['morning', 'hapenning'])  # {'hapenning'}
for word in misspelled:
    spell.correction(word)  # 'happening'
```

If using a set of long words that is taking a long time to process corrections then the Levenshtein distance can be set to 1. The default, is 2.

```
from spellchecker import SpellChecker

spell = SpellChecker(distance=1)  # set the Levenshtein Distance parameter

# do additional work

# now for shorter words, we can revert to Levenshtein Distance of 2!
spell.distance = 2
```

Or if the word identified as the likeliest is not correct, a list of candidates can also be pulled:

```
from spellchecker import SpellChecker

spell = SpellChecker()

misspelled = spell.unknown(['morning', 'hapenning'])  # {'hapenning'}
for word in misspelled:
    spell.correction(word)  # {'penning', 'happening', 'henning'}
```

7.1.3 Changing Language

To set the language of the dictionary to load, one must set the language parameter on initialization.

```
from spellchecker import SpellChecker

spell = SpellChecker(language='es')  # Spanish dictionary
print(spell['mañana'])
```

Multiple Languages

If you would like to check multiple default languages, it is possible to pass a list of language identifiers to the constructor to load each:

7.1.4 Adding and Removing Terms from a Dictionary

There are several ways to add additional terms to your word frequency dictionary including by filepath, string of text, or by a list of words.

To load a pre-defined dictionary file (either as a json file or a gzipped json file):

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell.word_frequency.load_dictionary('./path-to-my-word-frequency.json')
```

To load a text document that will be parsed into individual words and each word added to the frequency list:

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell.word_frequency.load_text_file('./path-to-my-text-doc.txt')
```

To load plain text from input or another source:

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell.word_frequency.load_text('Text to be parsed and added to the system')
```

Or update using a list of words:

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell.word_frequency.load_words(['Text', 'to', 'be','added', 'to', 'the', 'system'])
```

Or add a single word:

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell.word_frequency.add('Text')
```

Removing words is as simple as adding words:

```
from spellchecker import SpellChecker

spell = SpellChecker()
spell.word_frequency.remove_words(['Text', 'to', 'be','removed', 'from', 'the',
↪'system'])

# or remove a single word
spell.word_frequency.remove('meh')
```

7.1.5 Iterating Over a Dictionary

Iterating over the dictionary is as easy as writing a simple for loop:

```
from spellchecker import SpellChecker

spell = SpellChecker()

for word in spell:
    print("{}: {}".format(word, spell[word]))
```

The iterator returns the word. To get the number of times that the word is found in the WordFrequency object one can use a simple lookup.

7.1.6 How to Build a New Dictionary

Building a custom or new language dictionary is relatively straight forward. To begin, you will need to have either a word frequency list or text files that represent the usage of the terms. Since `pyspellchecker` uses word frequency, it is better to have the most common words have higher frequencies!

Once you have the corpus, code similar to the following should build out the dictionary:

```
from spellchecker import SpellChecker

# turn off loading a built language dictionary, case sensitive on (if desired)
spell = SpellChecker(language=None, case_sensitive=True)

# if you have a dictionary...
spell.word_frequency.load_dictionary('./path-to-my-json-dictionary.json')

# or... if you have text
spell.word_frequency.load_text_file('./path-to-my-text-doc.txt')

# export it out for later use!
spell.export('my_custom_dictionary.gz', gzipped=True)
```

It is also possible to build a dictionary from other sources outside of `pyspellchecker`, it requires that the data be in the following format and saved as a json object:

```
{
    "a": 1,
    "b": 2,
    "apple": 45,
    "bike": 60
}
```

Note that the data does not need to be sorted!

7.1.7 A quick, command line spell checking program

Setting up a quick and easy command line program using `pyspellchecker` is straight forward:

```
from spellchecker import SpellChecker

# could add command line arguments to set the parameters of the spell
# check class; setup what type of information to present back, etc.
spell = SpellChecker()

print("To exit, hit return without input!")
```

(continues on next page)

(continued from previous page)

```
while True:
    word = input('Input a word to spell check: ')
    if word == '':  # not sure, but need a way to kill the program...
        break
    word = word.lower()
    if word in spell:
        print("'{}' is spelled correctly!".format(word))
    else:
        cor = spell.correction(word)
        print("The best spelling for '{}' is '{}'".format(word, cor))

        print("If that is not enough; here are all possible candidate words:")
        print(spell.candidates(word))
```

7.1.8 Using with PyInstaller

It is possible to use `pyspellchecker` with tools such as PyInstaller to add spell-checking to your executable program. To do so, you will need to add the required dictionaries to the executable.

You will need to add the files to a folder in your executable called **spellchecker/resources/** to match the location that `pyspellchecker` checks for the supported dictionaries.

```
pyinstaller --add-binary="spellchecker/resources/en.json.gz:spellchecker/resources"␣
↪my_prog.py
```

On windows one should use a semi-colon instead of the colon:

```
pyinstaller --add-binary="spellchecker/resources/en.json.gz;spellchecker/resources"␣
↪my_prog.py
```

7.2 pyspellchecker API

Here you can find the full developer API for the pyspellchecker project. pyspellchecker provides a library for determining if a word is misspelled and what the likely correct spelling would be based on word frequency.

7.2.1 SpellChecker

class spellchecker.**SpellChecker**(*language: Union[str, Iterable[str]] = 'en', local_dictionary: Union[pathlib.Path, str, None] = None, distance: int = 2, tokenizer: Optional[Callable[[str], Iterable[str]]] = None, case_sensitive: bool = False*)

The SpellChecker class encapsulates the basics needed to accomplish a simple spell checking algorithm. It is based on the work by Peter Norvig (https://norvig.com/spell-correct.html)

Parameters

- **language** (*str*) – The language of the dictionary to load or None for no dictionary. Supported languages are *en*, *es*, *de*, *fr*, *pt*, *ru*, *lv*, and *eu*. Defaults to *en*. A list of languages may be provided and all languages will be loaded.
- **local_dictionary** (*str*) – The path to a locally stored word frequency dictionary; if provided, no language will be loaded

- **distance** (*int*) – The edit distance to use. Defaults to 2.
- **case_sensitive** (*bool*) – Flag to use a case sensitive dictionary or not, only available when not using a language dictionary.

Note: Using a case sensitive dictionary can be slow to correct words.

---

candidates (*word: Union[str, bytes]*) → Optional[Set[str]]
Generate possible spelling corrections for the provided word up to an edit distance of two, if and only when needed

Parameters **word** (*str*) – The word for which to calculate candidate spellings

Returns The set of words that are possible candidates or None if there are no candidates

Return type set

correction (*word: Union[str, bytes]*) → Optional[str]
The most probable correct spelling for the word

Parameters **word** (*str*) – The word to correct

Returns The most likely candidate or None if no correction is present

Return type str

distance
The maximum edit distance to calculate

Note: Valid values are 1 or 2; if an invalid value is passed, defaults to 2

Type int

edit_distance_1 (*word: Union[str, bytes]*) → Set[str]
Compute all strings that are one edit away from *word* using only the letters in the corpus

Parameters **word** (*str*) – The word for which to calculate the edit distance

Returns The set of strings that are edit distance one from the provided word

Return type set

edit_distance_2 (*word: Union[str, bytes]*) → List[str]
Compute all strings that are two edits away from *word* using only the letters in the corpus

Parameters **word** (*str*) – The word for which to calculate the edit distance

Returns The set of strings that are edit distance two from the provided word

Return type set

export (*filepath: Union[pathlib.Path, str], encoding: str = 'utf-8', gzipped: bool = True*) → None
Export the word frequency list for import in the future

Parameters

- **filepath** (*str*) – The filepath to the exported dictionary
- **encoding** (*str*) – The encoding of the resulting output
- **gzipped** (*bool*) – Whether to gzip the dictionary or not

known (*words: Iterable[Union[str, bytes]]*) → Set[str]
The subset of *words* that appear in the dictionary of words

Parameters **words** (*list*) – List of words to determine which are in the corpus

Returns The set of those words from the input that are in the corpus

Return type set

classmethod languages () → Iterable[str]
list: A list of all official languages supported by the library

split_words (*text: Union[str, bytes]*) → Iterable[str]
Split text into individual *words* using either a simple whitespace regex or the passed in tokenizer

Parameters **text** (*str*) – The text to split into individual words

Returns A listing of all words in the provided text

Return type list(str)

unknown (*words: Iterable[Union[str, bytes]]*) → Set[str]
The subset of *words* that do not appear in the dictionary

Parameters **words** (*list*) – List of words to determine which are not in the corpus

Returns The set of those words from the input that are not in the corpus

Return type set

word_frequency
An encapsulation of the word frequency *dictionary*

Note: Not settable

Type *WordFrequency*

word_usage_frequency (*word: Union[str, bytes], total_words: Optional[int] = None*) → float
Calculate the frequency to the *word* provided as seen across the entire dictionary

Parameters

- **word** (*str*) – The word for which the word probability is calculated
- **total_words** (*int*) – The total number of words to use in the calculation; use the default for using the whole word frequency

Returns The probability that the word is the correct word

Return type float

7.2.2 WordFrequency

class spellchecker.**WordFrequency** (*tokenizer=None*, *case_sensitive=False*)
Store the *dictionary* as a word frequency list while allowing for different methods to load the data and update over time

add (*word: Union[str, bytes], val: int = 1*) → None
Add a word to the word frequency list

Parameters

- **word** (*str*) – The word to add
- **val** (*int*) – The number of times to insert the word

dictionary
A counting dictionary of all words in the corpus and the number of times each has been seen

Note: Not settable

Type Counter

items() → Generator[Tuple[str, int], None, None]
Iterator over the words in the dictionary

Yields *str* – The next word in the dictionary int: The number of instances in the dictionary

Note: This is the same as *dict.items()*

keys() → Generator[str, None, None]
Iterator over the key of the dictionary

Yields *str* – The next key in the dictionary

Note: This is the same as *spellchecker.words()*

letters
The listing of all letters found within the corpus

Note: Not settable

Type set

load_dictionary(*filename: Union[pathlib.Path, str], encoding: str = 'utf-8'*) → None
Load in a pre-built word frequency list

Parameters

- **filename** (*str*) – The filepath to the json (optionally gzipped) file to be loaded
- **encoding** (*str*) – The encoding of the dictionary

load_json(*data: Dict[str, int]*) → None
Load in a pre-built word frequency list

Parameters **data** (*dict*) – The dictionary to be loaded

load_text(*text: Union[str, bytes], tokenizer: Optional[Callable[[str], Iterable[str]]] = None*) → None
Load text from which to generate a word frequency list

Parameters

- **text** (*str*) – The text to be loaded
- **tokenizer** (*function*) – The function to use to tokenize a string

load_text_file(*filename: Union[pathlib.Path, str], encoding: str = 'utf-8', tokenizer: Optional[Callable[[str], Iterable[str]]] = None*) → None
Load in a text file from which to generate a word frequency list

Parameters

- **filename** (*str*) – The filepath to the text file to be loaded
- **encoding** (*str*) – The encoding of the text file
- **tokenizer** (*function*) – The function to use to tokenize a string

load_words(*words: Iterable[Union[str, bytes]]*) → None
Load a list of words from which to generate a word frequency list

Parameters **words** (*list*) – The list of words to be loaded

longest_word_length
The longest word length in the dictionary

Note: Not settable

Type int

pop(*key: Union[str, bytes], default: Optional[int] = None*) → int
Remove the key and return the associated value or default if not found

Parameters

- **key** (*str*) – The key to remove
- **default** (*obj*) – The value to return if key is not present

remove(*word: Union[str, bytes]*) → None
Remove a word from the word frequency list

Parameters **word** (*str*) – The word to remove

remove_by_threshold(*threshold: int = 5*) → None
Remove all words at, or below, the provided threshold

Parameters **threshold** (*int*) – The threshold at which a word is to be removed

remove_words(*words: Iterable[Union[str, bytes]]*) → None
Remove a list of words from the word frequency list

Parameters **words** (*list*) – The list of words to remove

tokenize(*text: Union[str, bytes]*) → Generator[str, None, None]
Tokenize the provided string object into individual words

Parameters **text** (*str*) – The string object to tokenize

Yields *str* – The next *word* in the tokenized string

Note: This is the same as the *spellchecker.split_words()* unless a tokenizer function was provided.

---

total_words
The sum of all word occurrences in the word frequency dictionary

Note: Not settable

Type int

unique_words
The total number of unique words in the word frequency list

Note: Not settable

Type int

words() → Generator[str, None, None]
Iterator over the words in the dictionary

Yields *str* – The next word in the dictionary

Note: This is the same as *spellchecker.keys()*

CHAPTER 8

Additional Information

- genindex
- modindex
- search

Index