lunr.py

Yeray Diaz Diaz

Apr 22, 2021

CONTENTS

A Python implementation of Lunr.js by Oliver Nightingale.

> A bit like Solr, but much smaller and not as bright.

This Python version of Lunr.js aims to bring the simple and powerful full text search capabilities into Python guaranteeing results as close as the original implementation as possible.

CHAPTER

ONE

WHAT DOES THIS EVEN DO?

Lunr is a simple full text search solution for situations where deploying a full scale solution like Elasticsearch isn't possible, viable or you're simply prototyping. Lunr parses a set of documents and creates an inverted index for quick full text searches in the same way other more complicated solution.

The trade-off is that Lunr keeps the inverted index in memory and requires you to recreate or read the index at the start of your application.

CHAPTER

TWO

INTEROPERABILITY WITH LUNR.JS

A core objective of Lunr.py is to *provide interoperability with the JavaScript version*.

An example can be found in the MkDocs documentation library. MkDocs produces a set of documents from the pages of the documentation and uses Lunr.js in the frontend to power its built-in searching engine. This set of documents is in the form of a JSON file which needs to be fetched and parsed by Lunr.js to create the inverted index at startup of your application.

While this is not a problem for most sites, depending on the size of your document set, this can take some time.

Lunr.py provides a backend solution, allowing you to parse the documents in Python of time and create a serialized Lunr.js index you can pass have the browser version read, minimizing start up time of your application.

Each version of lunr.py targets a specific version of lunr.js and produces the same results as it both in Python 2.7 and 3 for non-trivial corpus of documents.

Lunr.py also serializes `Index` instances respecting the `lunr-schema` which are consumable by Lunr.js and vice-versa.

CHAPTER

THREE

INSTALLATION

```
pip install lunr
```

An optional and experimental support for other languages thanks to the Natural Language Toolkit stemmers is also available via `pip install lunr[languages]`. The usage of the language feature is subject to NTLK corpus licensing clauses.

Please refer to the documentation page on languages for more information.

CHAPTER

FOUR

USAGE

First, you'll need a list of dicts representing the documents you want to search on. These documents must have a unique field which will serve as a reference and a series of fields you'd like to search on.

Lunr provides a convenience `lunr` function to quickly index this set of documents:

```
>>> from lunr import lunr
>>>
>>> documents = [{
...     'id': 'a',
...     'title': 'Mr. Green kills Colonel Mustard',
...     'body': 'Mr. Green killed Colonel Mustard in the study with the candlestick.',
... }, {
...     'id': 'b',
...     'title': 'Plumb waters plant',
...     'body': 'Professor Plumb has a green plant in his study',
... }]
>>> idx = lunr(
...     ref='id', fields=('title', 'body'), documents=documents
... )
>>> idx.search('kill')
[{'ref': 'a', 'score': 0.6931722372559913, 'match_data': <MatchData "kill">}]
>>> idx.search('study')
[{'ref': 'b', 'score': 0.23576799568081389, 'match_data': <MatchData "studi">}, {'ref
↪': 'a', 'score': 0.2236629211724517, 'match_data': <MatchData "studi">}]
```

4.1 Quick start

First, you'll need a list of dicts representing the documents you want to search on. These documents must have a unique field which will serve as a reference and a series of fields you'd like to search on.

```
>>> from lunr import lunr
>>>
>>> documents = [{
...:         'id': 'a',
...:         'title': 'Mr. Green kills Colonel Mustard',
...:         'body': """Mr. Green killed Colonel Mustard in the study with the
...: candlestick. Mr. Green is not a very nice fellow."""
...:     }, {
...:         'id': 'b',
...:         'title': 'Plumb waters plant',
...:         'body': 'Professor Plumb has a green and a yellow plant in his study',
```

(continues on next page)

(continued from previous page)

```
...:     }, {
...:         'id': 'c',
...:         'title': 'Scarlett helps Professor',
...:         'body': """Miss Scarlett watered Professor Plumbs green plant
...: while he was away on his murdering holiday.""",
...:     }]
```

Lunr provides a convenience `lunr` function to quickly index this set of documents:

```
>>> idx = lunr(
...     ref='id', fields=('title', 'body'), documents=documents
... )
```

For basic no-fuss searches just use the `search` on the index:

```
>>> idx.search('kill')
[{'ref': 'a', 'score': 0.6931722372559913, 'match_data': <MatchData "kill">}]
>>> idx.search('study')
[{'ref': 'b', 'score': 0.23576799568081389, 'match_data': <MatchData "studi">},
{'ref': 'a', 'score': 0.2236629211724517, 'match_data': <MatchData "studi">}]
```

4.1.1 Using query strings

The query string passed to `search` accepts multiple terms:

```
>>> idx.search('green plant')
[{'ref': 'b', 'score': 0.5023294192217546, 'match_data': <MatchData "green, plant">},
{'ref': 'a', 'score': 0.12544083739725947, 'match_data': <MatchData "green">},
{'ref': 'c', 'score': 0.07306110905506158, 'match_data': <MatchData "green, plant">}]
```

The index will search for `green` OR `plant`, a few things to note on the results:

- document `b` scores highest because `plant` appears in both fields and `green` appears in the body
- document `a` is second includes only `green` but in the title and the body twice
- document `c` includes both terms but only on one of the fields

Query strings support a variety of modifiers:

Wildcards

You can use `*` as a wildcard anywhere in your query string:

```
>>> idx.search('pl*')
[{'ref': 'b', 'score': 0.725901569004226, 'match_data': <MatchData "plumb, plant">},
{'ref': 'c', 'score': 0.0816178155209697, 'match_data': <MatchData "plumb, plant">}]
>>> idx.search('*llow')
[{'ref': 'b', 'score': 0.6210112024848421, 'match_data': <MatchData "yellow">},
{'ref': 'a', 'score': 0.30426104537491444, 'match_data': <MatchData "fellow">}]
```

Note that, when using wildcards, no stemming is performed in the search terms.

Fields

Prefixing any search term with `<FIELD_NAME>:` allows you to specify which field a particular term should be searched for:

```
>>> idx.search('title:green title:plant')
[{'ref': 'b', 'score': 0.18604713274256787, 'match_data': <MatchData "plant">},
{'ref': 'a', 'score': 0.07902963505882092, 'match_data': <MatchData "green">}]
```

Note the difference with the example above, document `c` is no longer in the results.

Specifying an unindexed field will raise an exception:

```
>>> idx.search('foo:green')
Traceback (most recent call last):
...
lunr.exceptions.QueryParseError: Unrecognized field "foo", possible fields title, body
```

You can combine this with wildcards:

```
>>> idx.search('body:mu*')
[{'ref': 'c', 'score': 0.3072276611029057, 'match_data': <MatchData "murder">},
{'ref': 'a', 'score': 0.14581429988419872, 'match_data': <MatchData "mustard">}]
```

Boosts

When searching for several terms you can use boosting to give more importance to the each term:

```
>>> idx.search('green plant^10')
[{'ref': 'b', 'score': 0.831629678987025, 'match_data': <MatchData "green, plant">},
{'ref': 'c', 'score': 0.06360184858161157, 'match_data': <MatchData "green, plant">},
{'ref': 'a', 'score': 0.01756105367777591, 'match_data': <MatchData "green">}]
```

Note how document `c` now scores higher because of the boosting on the term `plant`. The `10` represents a multiplier on the relative score for the term and must be positive integers.

Fuzzy matches

You can also use fuzzy matching for terms that are likely to be misspelled:

```
>>> idx.search('yellow~1')
[{'ref': 'b', 'score': 0.621155860224936, 'match_data': <MatchData "yellow">},
{'ref': 'a', 'score': 0.3040972809936496, 'match_data': <MatchData "fellow">}]
```

The positive integer after `~` represents the edit distance, in this case 1 character, either by addition, removal or transposition.

Term presence (new in 0.3.0)

As mentioned above, Lunr defaults to searching for logical OR on terms, but it is possible to specify the presence of each term in matching documents. The default OR behaviour is represented by the term's presence being *optional* in a matching document, to specify that a term must be present in matching document the term must be prefixed with a +. On the other hand to specify that a term must *not* be included in a matching document the term must be prefixed with a −.

The below example searches for documents that must contain "green", might contain "plant" and must not contain "study":

```
>>> idx.search("+green plant -study")
[{'ref': 'c',
  'score': 0.08090317236904906,
  'match_data': <MatchData "green,plant">}]
```

Contrast this with the default behaviour:

```
>>> idx.search('green plant study')
[{'ref': 'b',
  'score': 0.5178296383103647,
  'match_data': <MatchData "green,plant,studi">},
 {'ref': 'a',
  'score': 0.22147889214939157,
  'match_data': <MatchData "green,studi">},
 {'ref': 'c',
  'score': 0.06605716362553504,
  'match_data': <MatchData "green,plant">}]
```

To simulate a logical AND search of "green AND plant" mark both terms as required:

```
>>> idx.search('+yellow +plant')
[{'ref': 'b',
  'score': 0.8915374700737615,
  'match_data': <MatchData "plant,yellow">}]
```

As opposed to the default:

```
>>> idx.search('yellow plant')
[{'ref': 'b',
  'score': 0.8915374700737615,
  'match_data': <MatchData "plant,yellow">},
 {'ref': 'c',
  'score': 0.045333674172311975,
  'match_data': <MatchData "plant">}]
```

Note presence can also be combined with any of the other modifiers described above.

4.2 Building indices

We briefly skimmed over creating indices in Lunr in the *searching* section, let's go into more detail around what we need to build a Lunr index.

4.2.1 The `lunr` function

The main entry point to Lunr is the `lunr` function. It provides a simple way to create an index, define fields we're interested in and start indexing a corpus of documents.

We do that simply by providing:

- A `ref` string specifying the field in the documents that should be used as a key for each document.
- A `fields` list, which defines the fields in the documents that should be added to the index.
- A `documents` list, including a set of dictionaries representing the documents we want to index.

And that's it. The `lunr` function will create an index, configure it, add the documents and return the `lunr.Index` for you to start searching.

4.2.2 Build time boosts

New in version 0.4.0

Lunr also provides some very useful functionality for boosting at index building time. There are two types of boosts you can include: field boosts and document boosts.

Field boosts

Field boosts let Lunr know that, when searching, we care more about some fields than others, a typical example is adding a boost on the *title* of our documents so when searching for a term, if it is found in the title, the document will score higher.

To include a field boost we use the `fields` argument of the `lunr` function, instead of passing a list of strings as usual, we pass a list of dictionaries with two keys:

- `field_name` whose value will be the name of the field in the documents we want to index.
- `boost` an integer to be multiplied to the score when a match is found on this field.

For example:

```
>>> from lunr import lunr
>>> documents = [{
...:         'id': 'a',
...:         'title': 'Mr. Green kills Colonel Mustard',
...:         'body': """Mr. Green killed Colonel Mustard in the study with the
...: candlestick. Mr. Green is not a very nice fellow."""
...:     }, {
...:         'id': 'b',
...:         'title': 'Plumb waters plant',
...:         'body': 'Professor Plumb has a green and a yellow plant in his study',
...:     }, {
...:         'id': 'c',
...:         'title': 'Scarlett helps Professor',
```

(continues on next page)

(continued from previous page)

```
...:         'body': """Miss Scarlett watered Professor Plumbs green plant
...: while he was away on his murdering holiday.""",
...:     }]
>>> idx = lunr(
...:     ref='id',
...:     fields=[dict(field_name='title', boost=10), 'body'],
...:     documents=documents
...: )
```

Note how we're passing a dictionary only for `title`, `body` will have a neutral value for `boost`.

```
>>> idx.search('plumb')
[{'match_data': <MatchData "plumb">, 'ref': 'b', 'score': 1.599},
 {'match_data': <MatchData "plumb">, 'ref': 'c', 'score': 0.13}]
```

Note how the score for document `b` is much higher thanks to our field boost.

Document boosts

Document boosts let Lunr know that some documents are more important than others, for example we would like an FAQ page to show up higher in searches.

In Lunr we do this via the `documents` argument to the `lunr` function, instead of passing a list of dictionaries we pass a 2-tuple (or list) with the document dictionary as a first item and another dictionary as a second item. This second dictionary must have a single `boost` key with an integer to be applied to any matches on this particular document.

```
documents = [
    {
        'id': 'a',
        'title': 'Mr. Green kills Colonel Mustard',
        'body': """Mr. Green killed Colonel Mustard in the study with the
candlestick. Mr. Green is not a very nice fellow."""
    }, {
            'id': 'b',
            'title': 'Plumb waters plant',
            'body': 'Professor Plumb has a green and a yellow plant in his study',
    }, (
        {
            'id': 'c',
            'title': 'Scarlett helps Professor',
            'body': """Miss Scarlett watered Professor Plumbs green plant
    while he was away on his murdering holiday.""",
        }, {
            'boost': 10
        }
    )]
```

Note how the third member of a list is a tuple, now if we pass these documents to the `lunr` function and perform a search:

```
>>> idx = lunr(ref='id', fields=('title', 'body'), documents=documents)
>>> idx.search('plumb')
[{'match_data': <MatchData "plumb">, 'ref': 'c', 'score': 1.297},
 {'match_data': <MatchData "plumb">, 'ref': 'b', 'score': 0.3}]
```

The score for `c` is now higher than `b` even though there are less matches, thanks to our document boost.

4.2.3 Field extractors

Up until now we've been working with fairly simple documents, but what if you have large nested documents and only want to index parts of them?

For this Lunr provides *field extractors*, which are simply callables that Lunr can use to fetch the field in the document you want to index. If you do not provide it, as we've been doing, Lunr assumes there's a key matching the field name, i.e. `title` or `body`.

To pass a field extractor to Lunr we, once again, use the `fields` argument to the `lunr` function. Similarly to what we did to define field boosts we pass a list of dictionaries, but this time we add an `extractor` key whose value is a callable with a single argument, the document being processed. Lunr will call the extractor when fetching the indexed field and will use its result in our index.

Imagine our documents have a slightly different form where the reference is at the top level but our fields are nested under a `content` key:

```
documents = [{
    'id': 'a',
    'content': {
        'title': 'Mr. Green kills Colonel Mustard',
        'body': """Mr. Green killed Colonel Mustard in the study with the
candlestick. Mr. Green is not a very nice fellow."""
    }
}, {
    'id': 'b',
    'content': {
        'title': 'Plumb waters plant',
        'body': 'Professor Plumb has a green and a yellow plant in his study',
    }
}, {
    'id': 'c',
    'content': {
        'title': 'Scarlett helps Professor',
        'body': """Miss Scarlett watered Professor Plumbs green plant
    while he was away on his murdering holiday.""",
    }
}]
```

To work around this we simply need to add field extractors, which are simply callables that take a document as an argument and return the content of the field, in this case a simple `lambda` will do:

```
>>> idx = lunr(
...     ref='id',
...     fields=[
...         dict(field_name='title', extractor=lambda d: d['content']['title']),
...         dict(field_name='body', extractor=lambda d: d['content']['body'])
...     ],
...     documents=documents)
```

We can now search the index as usual:

```
>>> idx.search('plumb')
[{'ref': 'b', 'score': 0.3, 'match_data': <MatchData "plumb">}
 {'ref': 'c', 'score': 0.13, 'match_data': <MatchData "plumb">}]
```

4.3 Language support

Lunr includes optional and experimental support for languages other than English via the Natural Language Toolkit. To install Lunr with this feature use `pip install lunr[languages]`.

The currently supported languages are:

- Arabic
- Danish
- Dutch
- English
- Finnish
- French
- German
- Hungarian
- Italian
- Norwegian
- Portuguese
- Romanian
- Russian
- Spanish
- Swedish

```
>>> documents = [
...   {
...     "id": "a",
...     "text": (
...         "Este es un ejemplo inventado de lo que sería un documento en el "
...         "idioma que se más se habla en España."),
...     "title": "Ejemplo de documento en español"
...   },
...   {
...     "id": "b",
...     "text": (
...         "Según un estudio que me acabo de inventar porque soy un experto en"
...         "idiomas que se hablan en España."),
...     "title": "Español es el tercer idioma más hablado del mundo"
...   },
... ]
```

New in 0.5.1: the `lunr` function now accepts more than one language

Simply define specify one or more ISO-639-1 codes for the language(s) of your documents in the `languages` parameter to the `lunr` function.

!!! Note In versions of Lunr prior to 0.5.0 the parameter's name is `language` and accepted a single string.

If you have a single language you can pass the language code in `languages`:

```
>>> from lunr import lunr
>>> idx = lunr('id', ['title', 'text'], documents, languages='es')
>>> idx.search('inventando')
[{'ref': 'a', 'score': 0.130, 'match_data': <MatchData "invent">},
{'ref': 'b', 'score': 0.089, 'match_data': <MatchData "invent">}]
```

!!! Note In order to construct stemmers, trimmers and stop word filters Lunr imports corpus data from NLTK which fetches data from Github and caches it in your home directory under `nltk_data` by default. You may see some logging indicating such activity during the creation of the index.

If you have documents in multiple language pass a list of language codes:

```
>>> documents.append({
     "id": "c",
     "text": "Let's say you also have documents written in English",
     "title": "A document in English"
 })
>>> idx = lunr('id', ['title', 'text'], documents, languages=['es', 'en'])
>>> idx.search('english')
[{'ref': 'c', 'score': 1.106, 'match_data': <MatchData "english">}]
```

4.3.1 Notes on language support

- Using multiple languages means the terms will be stemmed once per language. This can yield unexpected results.
- Compatibility with Lunr.js is ensured for languages that supported by both platforms, however results might differ slightly.
  - Languages supported by Lunr.js but not by Lunr.py:
    - Thai
    - Japanese
    - Turkish
  - Languages supported by Lunr.py but not Lunr.js:
    - Arabic
- The usage of the language feature is subject to NTLK corpus licensing clauses

4.4 Interoperability with Lunr.js

A key goal of Lunr.py is interoperability with Lunr.js: building an index with Lunr.py and being able to read it using Lunr.js without having to build it on the client on each visit.

The key step in this process is index serialization, which is possible thanks to `lunr-schema`.

The serialization process in Lunr.py consist on calling `Index.serialize`, here is a complete example with the data from the *introduction*:

```
>>> import json
>>> from lunr import lunr
>>> documents = [{
...         'id': 'a',
```

(continues on next page)

(continued from previous page)

```
...:         'title': 'Mr. Green kills Colonel Mustard',
...:         'body': """Mr. Green killed Colonel Mustard in the study with the
...: candlestick. Mr. Green is not a very nice fellow."""
...:     }, {
...:         'id': 'b',
...:         'title': 'Plumb waters plant',
...:         'body': 'Professor Plumb has a green and a yellow plant in his study',
...:     }, {
...:         'id': 'c',
...:         'title': 'Scarlett helps Professor',
...:         'body': """Miss Scarlett watered Professor Plumbs green plant
...: while he was away on his murdering holiday.""",
...:     }]
>>> idx = lunr(
...:     ref='id',
...:     fields=[dict(field_name='title', boost=10), 'body'],
...:     documents=documents
...: )
>>> serialized_idx = idx.serialize()
>>> with open('idx.json', 'w') as fd:
...:     fd.write(json.dump(serialized_idx))
```

As you can see `serialize` will produce a JSON friendly dict you can write to disk and read from Lunr.js. The following snippet shows how to read the index using Node.js:

```
> const fs = require('fs')
> const lunr = require('lunr')
> const serializedIndex = JSON.parse(fs.readFileSync('idx.json'))
> let idx = lunr.Index.load(serializedIndex)
> idx.search('plant')
[
  {
    ref: 'b',
    score: 1.599,
    matchData: { metadata: [Object: null prototype] }
  },
  {
    ref: 'c',
    score: 0.13,
    matchData: { metadata: [Object: null prototype] }
  }
]
```

!!! Note The search will only the *references* of the matching documents. It is up to you to keep mapping of the documents in memory to be able show richer results which means in a web environment you will need to serve *two* files, one for the index and another the collection of documents.

4.4.1 Loading a serialized index

You can also do the reverse operation of reading a serialized index produced by Lunr.py or Lunr.js using the `Index.load` class method:

```
>>> import json
>>> from lunr.index import Index
>>> with open("idx.json") as fd:
...     serialized_idx = json.loads(fd.read())
...
>>> idx = Index.load(serialized_idx)
>>> idx.search("plant")
[{'ref': 'b', 'score': 1.599, 'match_data': <MatchData "plant">}, {'ref': 'c', 'score
↪': 0.13, 'match_data': <MatchData "plant">}]
```

4.4.2 Language support

Lunr.js uses the `lunr-languages` package, a community driven collection of stemmers and trimmers for many languages.

Porting each of those into Python was not feasible so Lunr.py uses NTLK for language support and will configure the serialized index as expected by Lunr.js to ensure compatibility.

However, this produces differences in scoring when loading indices from Lunr.py into Lunr.js larger than those observed using the base english implementation, due to inherent differences in the implementation of said stemmers and trimmers.

4.5 Changelog

4.5.1 0.6.0 (2021-04-22)

- Add index customisation, enabling build and search pipeline tweaks as well as meta-data whitelisting.

4.5.2 0.5.9 (2021-01-10)

- Compatibility with Lunr.js 2.3.9:
  - Fix bug where clause matches are incorrectly initialized to a complete set.
- Add support for Python 3.9
- Drop support for Python 3.5

4.5.3 0.5.8 (2020-04-16)

- Fix installing ntlk in 2.7 without `languages` extra.
- Optimize regexes and avoid usage by default.

Deprecation warning

- 0.5.8 will be the last release to support Python 2.7.

4.5.4 0.5.7 (2020-04-14)

- Prevent installing an unsupported version of NLTK in Python 2.7.

4.5.5 0.5.6 (2019-11-17)

- Support for Python 3.8
- Compatibility with Lunr.js 2.3.8:
  - Fix bug where leading white space would cause token position metadata to be reported incorrectly.

4.5.6 0.5.5 (2019-04-28)

- Compatibility with Lunr.js 2.3.6:
  - Fix bug with fuzzy matching that meant deletions at the end of a word would not match.

4.5.7 0.5.4 (2018-11-10)

- Compatibility with Lunr.js 2.3.5:
  - Fix bug on fuzzy matching ignoring matches on insertions at the end of the word.

4.5.8 0.5.3 (2018-09-08)

- Performance improvements on indexing
- Compatibility with Lunr.js 2.3.3:
  - Fixes catastrophic backtracking on leading wildcards

4.5.9 0.5.2 (2018-08-25)

- Fix Python 2.7 support

4.5.10 0.5.1 (2018-08-25)

- Added multilanguage support
- Improved language support

Deprecation warning

- The `language` argument to the `lunr` has been renamed to `languages` to accomodate for multilanguage support. The `languages` argument accepts a string or an iterable of ISO-639-1 languages codes. If you're calling `lunr` with keyword arguments please update such calls accordingly.

4.5.11 0.4.3 (2018-08-18)

- Target Lunr.js v2.3.2

4.5.12 0.4.2 (2018-07-28)

- Target Lunr.js v2.3.1
- Fix crash when using non-string document references.

4.5.13 0.4.1 (2018-07-07)

- Added support for Python 3.7

4.5.14 0.4.0 (2018-06-25)

- Compatibility with Lunr.js v2.3.0. Including:
  - Add support for build time field and document boosts.
  - Add support for indexing nested document fields using field extractors.
  - Prevent usage of problematic characters in field names

4.5.15 0.3.0 (2018-06-03)

- Compatibility with Lunr.js v2.2.1. Including:
  - Add support for queries with term presence, e.g. required terms and prohibited terms.
  - Add support for using the output of `lunr.Tokenizer` directly with `lunr.Query.term`.
  - Add field name metadata to tokens in build and search pipelines.

4.5.16 0.2.3 (2018-05-19)

- Compatibility with Lunr.js v2.1.6

4.5.17 0.2.2 (2018-05-15)

- Fix bug on whitelisting metadata in Builder.

4.5.18 0.2.1 (2018-04-21)

- Refactor of multilanguage support.

4.5.19 0.2.0 (2018-04-15)

- Experimental support for languages via NLTK, currently supported languages are arabic, danish, dutch, english, finnish, french, german, hungarian, italian, norwegian, portuguese, romanian, russian, spanish and swedish. Note compatibility with Lunr.js and lunr-languages is reduced.

4.5.20 0.1.2 (2018-03-17)

- Add serialization tests passing serialized index from Python to JS and producing same results.
- Added `Index.create_query` returning a preinitialized `Query` with the index's fields or a subset of them.
- `Index.search` does not accept a callback function, instead expects a `Query` object the user should preconfigure first.
- Various docstring and repr changes.

4.5.21 0.1.1a1 (2018-03-01)

- Initial release

4.6 Customisation

Lunr.py ships with some sensible defaults to create indexes and search easily, but in some cases you may want to tweak how documents are indexed and search. You can do that in lunr.py by passing your own `Builder` instance to the `lunr` function.

4.6.1 Pipeline functions

When the builder processes your documents it splits (tokenises) the text, and applies a series of functions to each token. These are called pipeline functions.

The builder includes two pipelines, indexing and searching.

If you want to change the way lunr.py indexes the documents you'll need to change the indexing pipeline.

For example, say you wanted to support the American and British way of spelling certain words, you could use a normalisation pipeline function to force one token into the other:

```
from lunr import lunr, get_default_builder
import lunr.pipeline.Pipeline

documents = [...]

builder = get_default_builder()
def normalise_spelling(token, i, tokens) {
    if str(token) == "gray":
        return token.update(lambda: "grey")
    else:
        return token

lunr.pipeline.Pipeline.register_function(normalise_spelling)
builder.pipeline.add(normalise_spelling)

idx = lunr(ref="id", fields=("title", "body"), documents=documents, builder=builder)
```

Note pipeline functions take the token being processed, its position in the token list, and the token list itself.

4.6.2 Token meta-data

Lunr.py `Token` instances include meta-data information which can be used in pipeline functions. This meta-data is not stored in the index by default, but it can be by adding it to the builder's `metadata_whitelist` property. This will include the meta-data in the search results:

```
from lunr import lunr, get_default_builder
import lunr.pipeline.Pipeline

builder = get_default_builder()

def token_length(token, i, tokens):
    token.metadata["token_length"] = len(str(token))
    return token

Pipeline.register_function(token_length)
builder.pipeline.add(token_length)
builder.metadata_whitelist.append("token_length")

idx = lunr("id", ("title", "body"), documents, builder=builder)

[result, _, _] = idx.search("green")
assert result["match_data"].metadata["green"]["title"]["token_length"] == [5]
assert result["match_data"].metadata["green"]["body"]["token_length"] == [5, 5]
```

4.6.3 Similarity tuning

The algorithm used by Lunr to calculate similarity between a query and a document can be tuned using two parameters. Lunr ships with sensible defaults, and these can be adjusted to provide the best results for a given collection of documents.

- **b**: This parameter controls the importance given to the length of a document and its fields. This value must be between 0 and 1, and by default it has a value of 0.75. Reducing this value reduces the effect of different length documents on a term's importance to that document.

- **k1**: This controls how quickly the boost given by a common word reaches saturation. Increasing it will slow down the rate of saturation and lower values result in quicker saturation. The default value is 1.2. If the collection of documents being indexed have high occurrences of words that are not covered by a stop word filter, these words can quickly dominate any similarity calculation. In these cases, this value can be reduced to get more balanced results.

These values can be changed in the builder:

```
from lunr import lunr, get_default_builder

builder = get_default_builder()
builder.k1(1.3)
builder.b(0)

idx = lunr("id", ("title", "body"), documents, builder=builder)
```