SimpleNeighbors Documentation

Release 0.1.0

Allison Parrish

Jan 13, 2020

Contents

Simple Neighbors is a clean and easy interface for performing nearest-neighbor lookups on items from a corpus. To install the package:

```
pip install simpleneighbors[annoy]
```

Here's a quick example, showing how to find the names of colors most similar to 'pink' in the xkcd colors list:

```
>>> from simpleneighbors import SimpleNeighbors
>>> import json
>>> color_data = json.load(open('xkcd.json'))['colors']
>>> hex2int = lambda s: [int(s[n:n+2], 16) for n in range(1,7,2)]
>>> colors = [(item['color'], hex2int(item['hex'])) for item in color_data]
>>> sim = SimpleNeighbors(3)
>>> sim.feed(colors)
>>> sim.build()
>>> list(sim.neighbors('pink', 5))
['pink', 'bubblegum pink', 'pale magenta', 'dark mauve', 'light plum']
```

For a more complete example, refer to my Understanding Word Vectors notebook, which shows how to use Simple Neighbors to perform similarity lookups on word vectors.

Read the complete Simple Neighbors documentation here: https://simpleneighbors.readthedocs.org.

CHAPTER 1

Why Simple Neighbors?

Approximate nearest-neighbor lookups are a quick way to find the items in your data set that are closest (or most similar to) any other item in your data, or an arbitrary point in the space that your data defines. Your data items might be colors in a (R, G, B) space, or sprites in a (X, Y) space, or word vectors in a 300-dimensional space.

You could always perform pairwise distance calculations to find nearest neighbors in your data, but for data of any appreciable size and complexity, this kind of calculation is unbearably slow. Simple Neighbors uses one of a handful of libraries behind the scenes to provide approximate nearest-neighbor lookups, which are ultimately a little less accurate than pairwise calculations but much, much faster.

The library also keeps track of your data, sparing you the extra step of mapping each item in your data to its integer index (at the potential cost of some redundancy in data storage, depending on your application).

I made Simple Neighbors because I use nearest neighbor lookups all the time and found myself writing and rewriting the same bits of wrapper code over and over again. I wanted to hide a little bit of the complexity of using these libraries to make it easier to build small prototypes and teach workshops using nearest-neighbor lookups.

CHAPTER 2

Multiple backend support

Simple Neighbors relies on the approximate nearest neighbor index implementations found in other libraries. By default, Simple Neighbors will choose the best backend based on the packages installed in your environment. (You can also specify which backend to use by hand, or create your own.)

Currently supported backend libraries include:

- `Annoy`: Erik Bernhardsson's Annoy library
- `Sklearn`: scikit-learn's NearestNeighbors
- `BruteForcePurePython`: Pure Python brute-force search (included in package)

When you install Simple Neighbors, you can direct `pip` to install the required packages for a given backend. For example, to install Simple Neighbors with Annoy:

```
pip install simpleneighbors[annoy]
```

Annoy is highly recommended! This is the preferred way to use Simple Neighbors.

To install Simple Neighbors alongside scikit-learn to use the `Sklearn` backend (which makes use of scikit-learn's *NearestNeighbors* class):

```
pip install simpleneighbors[sklearn]
```

If you can't install Annoy or scikit-learn on your platform, you can also use a pure Python backend:

```
pip install simpleneighbors[purepython]
```

Note that the pure Python version uses a brute force search and is therefore very slow. In general, it's not suitable for datasets with more than a few thousand items (or more than a handful of dimensions).

See the documentation for the `SimpleNeighbors` class for more information on specifying backends.

CHAPTER 3

Contents

3.1 Simple Neighbors API Reference

class `simpleneighbors.`**`SimpleNeighbors`**(*dims*, *metric='angular'*, *backend=None*)

A Simple Neighbors index.

This class wraps backend implementations of approximate nearest neighbors indexes with a user-friendly API. When you instantiate this class, it will automatically select a backend implementation based on packages installed in your environment. It is HIGHLY RECOMMENDED that you install Annoy (`pip install annoy`) to enable the Annoy backend! (The alternatives are slower and not as accurate.) Alternatively, you can specify a backend of your choosing with the `backend` parameter.

Specify the number of dimensions in your data (i.e., the length of the list or array you plan to provide for each item) and the distance metric you want to use. The default is `angular` distance, an approximation of cosine distance. This metric is supported by all backends, as is `euclidean` (for Euclidean distance). Both of these parameters are passed directly to the backend; see the backend documentation for more details.

Parameters

- **`dims`** – the number of dimensions in your data
- **`metric`** – the distance metric to use
- **`backend`** – the nearest neighbors backend to use (default is annoy)

`add_one`(*item*, *vector*)

Adds an item to the index.

You need to provide the item to add and a vector that corresponds to that item. (For example, if the item is the name of a color, the vector might be a (R, G, B) triplet corresponding to that color. If the item is a word, the vector might be a word2vec or GloVe vector corresponding to that word.

Items can be any hashable Python object. Vectors must be sequences of numbers. (Lists, tuples, and Numpy arrays should all be fine, for example.)

Note: If the index has already been built, you won't be able to add new items.

Parameters

- **item** – the item to add
- **vector** – the vector corresponding to that item

Returns None

build(*n=10*, *params=None*)

Build the index.

After adding all of your items, call this method to build the index. The meaning of parameter `n` is different for each backend implementation. For the Annoy backend, it specifies the number of trees in the underlying Annoy index (a higher number will take longer to build but provide more precision when querying). For the Sklearn backend, the number specifies the leaf size when building the ball tree. (The Brute Force Pure Python backend ignores this value entirely.)

After you call build, you'll no longer be able to add new items to the index.

Parameters

- **n** – backend-dependent (for Annoy: number of trees)
- **params** – dictionary with extra parameters to pass to backend

dist(*a*, *b*)

Returns the distance between two items.

Parameters

- **a** – first item
- **b** – second item

Returns distance between `a` and `b`

feed(*items*)

Add multiple items to the index.

Supply to this method a sequence of (item, vector) tuples (e.g., a list of tuples, a generator that produces tuples, etc.) and they'll all be added to the index. Great for adding multiple items to the index at once.

Items can be any hashable Python object. Vectors must be sequences of numbers. (Lists, tuples, and Numpy arrays should all be fine, for example.)

Parameters **items** – a sequence of (item, vector) tuples

Returns None

classmethod load(*prefix*)

Restores a previously-saved index.

This class method restores a previously-saved index using the specified file prefix.

Parameters **prefix** – prefix used when saving

Returns SimpleNeighbors object restored from specified files

nearest(*vec*, *n=12*)

Returns the items nearest to a given vector.

The specified vector must have the same number of dimensions as the number given when initializing the index. The nearest neighbor search is limited to the given number of items, and results are sorted in order of proximity.

```
>>> from simpleneighbors import SimpleNeighbors
>>> sim = SimpleNeighbors(2, 'euclidean')
>>> sim.feed([('a', (4, 5)),
...     ('b', (0, 3)),
...     ('c', (-2, 8)),
...     ('d', (2, -2))])
>>> sim.build()
>>> sim.nearest((1, -1), n=1)
['d']
```

Parameters

- **vec** – search vector
- **n** – number of results to return

Returns a list of items sorted in order of proximity

nearest_matching (*vec*, *n=12*, *check=lambda x: True*)

Returns the items nearest a given vector that pass a test.

This method looks for the items in the index nearest the given vector that meet a particular criterion. It tries to find at least `n` items, expanding the search as needed. (It may yield fewer than the desired number if enough items can't be found in the entire index.)

The function passed as `check` will be called with a single parameter: the item in question. It should return `True` if the item should be included in the results, and `False` otherwise.

This search process might be slow; in order to make it easier to display incremental results, this method returns a generator. You can easily get the results of this method as a list by enclosing your call inside the `list()` function.

```
>>> from simpleneighbors import SimpleNeighbors
>>> sim = SimpleNeighbors(2, 'euclidean')
>>> sim.feed([('a', (4, 5)),
...     ('b', (0, 3)),
...     ('c', (-2, 8)),
...     ('d', (2, -2))])
>>> sim.build()
>>> list(sim.nearest_matching((3.5, 4.5), n=1,
...     check=lambda x: ord(x) <= ord('b')))
['a']
```

Parameters

- **vec** – search vector
- **n** – number of items to return
- **check** – function to call on each item

Returns a generator yielding up to `n` items

neighbors (*item*, *n=12*)

Returns the items nearest another item in the index.

This method returns the items closest to a given item in the index in order of proximity, limiting results to the number specified. (It's just like `nearest()` except using the vector of an item already in the corpus.)

```
>>> from simpleneighbors import SimpleNeighbors
>>> sim = SimpleNeighbors(2, 'euclidean')
>>> sim.feed([('a', (4, 5)),
...     ('b', (0, 3)),
...     ('c', (-2, 8)),
...     ('d', (2, -2))])
>>> sim.build()
>>> sim.neighbors('b', n=3)
['b', 'a', 'c']
```

Parameters

- **`item`** – a data item in that has already been added to the index
- **`n`** – the number of items to return

Returns a list of items sorted in order of proximity

`neighbors_matching`(*item*, *n=12*, *check=None*)

Returns the items nearest an indexed item that pass a test.

This method is just like `nearest_matching()`, but finds items nearest a given item already in the index, instead of an arbitrary vector.

Parameters

- **`item`** – search item
- **`n`** – number of items to return
- **`check`** – function to call on each item

Returns a generator yielding up to `n` items

`save`(*prefix*)

Saves the index to disk.

This method saves the index to disk. Each backend manages serialization a little bit differently: consult the documentation and source code for more details. For example, because Annoy indexes can't be serialized with *pickle*, the Annoy backend's implementation produces two files: the serialized Annoy index, and a pickle with the other data from the object.

This method's parameter specifies the "prefix" to use for these files.

Parameters **`prefix`** – filename prefix for Annoy index and object data

Returns None

`vec`(*item*)

Returns the vector for an item.

This method returns the vector that was originally provided when indexing the specified item. (Depending on how it was originally specified, they may have been converted to a different data type; e.g., integer vectors are converted to floats.)

Parameters **`item`** – item to lookup

Returns vector for item

3.2 Credits and Acknowledgements

Lead developer: Allison Parrish <allison@decontextualize.com>.

3.3 History

3.3.1 0.1.0 (2020-01-12)

- Support for multiple backends. This was implemented primarily to ease installation for users who can't install Annoy (because of a lack of binary packaging for their platforms).

3.3.2 0.0.1 (2018-07-13)

- Initial release.

Python Module Index

Index