vecto Documentation

vecto authors

Jun 13, 2018

Contents

Vecto is an open-source Python library for working with vector space models (VSMs), including various word embeddings such as word2vec. Vecto can load various popular formats of VSMs and perform a set of basic operations like dimensionality reduction, seach for nearest neighbors etc. It includes a growing list of benchmarks with which VSMs are evaluated in most current research, and a few visualization tools. It also includes a growing list of modules for creating VSMs, both explicit and based on neural networks.

CHAPTER 1

Tutorial

1.1 Introduction to Vecto

This is the tutorial for Vecto. It describes:

- What it is, and why we are developing it.
- what you can do with Vecto.
- the roadmap of the project.

Both the library and the documentation are actively developed, check back for more! If you have questions, or would like to contribute, feel free to get in touch on github.

1.1.1 What is Vecto?

Vecto is an open-source Python library for working with vector space models (VSMs), including various word embeddings such as word2vec. Vecto can load various popular formats of VSMs and retrieve nearest neighbors of a given vector. It includes a growing list of benchmarks with which VSMs are evaluated in most current research, and a few visualization tools. It also includes a growing list of modules for creating VSMs, both explicit and based on neural networks.

1.1.2 Why do you bother?

There are a few other libraries for working with VSMs, including gensim and spacy. Vecto differs from them in that its primary goal is to facilitate pricipled, systematic research in providing **a framework for reproducible experiments** on VSMs.

From the academic perspective, this matters because this is the only way to understand more about what VSMs are and what kind of meaning representation they offer.

From the practical perspective, this matters because otherwise we can not tell which VSM would be the best to use for what task. Existing extrinsic evaluations of VSMs such as popular word similarity, relatedness, analogy and intrusion

tasks have methodological problems and do not correlate well with performance on all extrinsic tasks. Therefore basically to pick the best representation for a task you have to try different kinds of VSMs until you find the best-performing one.

Furthermore, there is the important and unpleasant part of parameter tuning and optimizing for a particular task. Levy et al. (2015) showed that the choice of hyperparameters may make more of a difference than the choice of model itself. Even more frustratingly, when you have a relatively comprehensive task covering a wide range of linguistic relations, you may find that the parameters beneficial to a part of the task are detrimental for another part (Gladkova et al. 2016).

The neural parts of Vecto is implemented in Chainer, a new deep learning framework that is friendly to high-performace multi-GPU environments. This should make Vecto useful in both academic and industrial settings.

1.2 Installing Vecto

1.2.1 System requirements

- Python 3.5 or later

1.2.2 Method 1: Pip-install

The latest stable version:

```
>>> pip3 install vecto
```

The latest development version:

```
>>> pip3 install git+https://github.com/vecto-ai/vecto.git
```

1.2.3 Method 2: Clone or download the github repo

You can avoid intalling vecto system-wide. Simply download and unpack the github repo into your project's working directory.

Either way, you can access the vecto's modules by issuing

```
>>> import vecto
```

at the beginning of your code.

1.3 The metadata

Vecto attempts to record and track as much information as possible about each embedding and each experiment you run. All the information about VSMs is stored in a *metadata.json* file in the same folder as the VSM itself.

Vecto can be used to work with VSMs that were trained elsewhere and may not come with any metadata. However, even in this case, we encourage the users to try and find out and record as much of the metadata as possible,as soon as possible. We have all been in the situation where, long after you have published a paper and forgotten all about that project, you need to reuse some code or repeat an experiment - and that it's nigh impossible, because the code is unreadabe, filenames are criptic, and filepaths are long gone.

Moreover, keeping track of the metadata is also something that would force the researchers to be more aware of all these different hidden variables in their experiments. That would (1) prevent them from misinterpreting the properties of their models, and (2) provide some ideas about what could be tweaked.

1.3.1 The corpus metadata

It all starts with the corpus. Actually, as many corpora as you like, since it is common practice to combine corpora to train a model (to increase the volume of data, to diversify it, or in fancy curriculum learning). Here is a sample metadata file you can use as a template to describe your corpus.

Vecto records the following metadata:

todo a page about domains

id An identifier of the corpus, unique in the collection.

size The size of the corpus (in tokens).

name The (preferably short) name of the corpus, often used to identify the models built from it.

description The freeform description of the corpus, such as the domains it covers.

source Where the corpus was obtained.

domain The list of the domains of the texts, such as **news**, **encyclopedia**, **fiction**, **medical**, **spoken**, or **web**. If the corpus covers only one domain, the list only contains one item; otherwise several can be listed. We suggest using **general** only for balanced, representative corpora such as BNC that make a conscious effort to represent different registers.

language A list containing the language codes for the corpus. There will be just one entry in case of monolingual corpora (e.g. _["en"]_), and for parallel or multilingual corpora there will be several (_["en", "de"]_).

encoding The encoding of the corpus files.

format The format of the corpus. Some frequent options include: one-corpus-per-line, one-sentence-per-line, one-paragraph-per-line, one-word-per-line, vertical-format

date The date when the corpus (or its text source) was published. It can be the date of a Wikipedia dump (e.g. _2018-07_), or the year when the paper presenting the resource came out (e.g. _2017_).

path The path to the local copy of the corpus files.

cite The bibtex entry for the paper presenting the resource, that should be referenced in subsequent work building on or using the resource. It should be bibtex rather than biblatex, as most NLP publishers have not made the switch yet.

pre-processing The pre-processing steps used in preparing this resource, described in freeform text.

cleanup Markup removal, format conversion, encoding, de-duplication (freeform description, URL or path to the pre-processing script)

lowercasing **True** if the corpus was lowercased, **False** otherwise.

tokenization The tokenizer that was used, if any (URL or path to the script, name, version).

lemmatization The lemmatizer that was used, if any (URL or path to the script, name, version).

stemming The stemmer that was used, if any (URL or path to the script, name, version).

POS_tagging The POS-tagger that was used, if any (URL or path to the script, name, version).

syntactic_parsing The syntactic parser that was used, if any (URL or path to the script, name, version).

Semantic_parsing The semantic parser that was used, if any (URL or path to the script, name, version).

Other_preprocessing Any other pre-processing that was performed, if any (URL or path to the script, name, version).

todo the format section should link to the input of embedding models

```
{
"class": "corpus",
"corpus_01":    {
                 "id": "",
                 "size": ,
                 "name": "",
                 "description": "",
                 "source": "",
                 "domain": "",
                 "language": ["english"],
                 "encoding": "",
                 "format": "",
                 "date": "",
                 "path": "",

                 "pre-processing": {
                                   "cleanup": "",
                                   "lowercasing": ,
                                   "tokenization": "",
                                   "lemmatization": "",
                                   "stemming": "",
                                   "POS_tagging": "",
                                   "syntactic_parsing": "",
                                   "semantic_parsing": "",
                                   "other_preprocessing": "",
                                   }
                 }
"corpus_02":    {
                 "id": "",
                 "size": ,
                 "name": "",
                 "description": "",
                 "source": "",
                 "domain": "",
                 "language": ["english"],
                 "encoding": "",
                 "format": "",
                 "date": "",
                 "path": "",

                 "pre-processing": {
                                   "cleanup": "",
                                   "lowercasing": ,
                                   "tokenization": "",
                                   "lemmatization": "",
                                   "stemming": "",
                                   "POS_tagging": "",
                                   "syntactic_parsing": "",
                                   "semantic_parsing": "",
                                   "other_preprocessing": ""
                                   }
                 }
}
```

1.3.2 The vocab metadata

There are two types of vocab files in Vecto. One is basically lists of the vocabulary of word embeddings. Sometimes they are stored separately from the numerical data as plain-text, one-word-per-line files (e.g. when the numerical data itself is stored in .npy format). Vecto expects such files to have a ".vocab" extension.

The other type of vocab is a tab-separated file structured as [WORD FREQUENCY].

The vocab files can have associated metadata as follows.

size The number of token types.

min_frequency The minimum frequency cut-off point.

timestamp When the vocab file was produced

filtering A freeform description of any filtering applied to the vocabulary, if any.

lib_version The version of Vecto with which a given vocab file was produced (generated automatically by Vecto).

system_info The system in which the vocab file was produced (generated automatically by Vecto).

source Includes the metadata of the source corpus from which the vocab file was produced, as described in *The corpus metadata* section.

todo link to the vocab filtering section, if any

todo filtered_by - text file, wordlist, dict with metadata

```
{
"class": "vocabulary",
"size": ,
"min_frequency": ,
"lowercasing": "",
"execution_time": "",
"timestamp": "",
"lib_version": "",
"system_info": "",
"filtered_by": {
                },
"source":    {
             }
}
```

1.3.3 The embeddings metadata

The metadata collected in training of embeddings is hard to standartize, because essentially it needs to describe all the parameters of a given model, and they differ across models. Therefore this section only provides a sample, and the full list of parameters (which correspond to metadata) can be found in descriptions of the implementations of different models in the library.

todo link to the library of embeddings

Some of the frequent parameters applicable to most-if-not-all models include:

model The name of the model, such as CBOW or GloVe.

window The window size

dimensionality The number of vector dimensions.

context The type of context, as described by Li et al. Four common combinations are **linear_unbound** (the bag-of-words symmetrical context, the most commonly used), **linear_bound** (linear context that takes word order into account), **deps_unbound** (the dependency-based context which takes into account all words in a syntactic relation to the target word), and **deps_boun** (a version of the latter which differentiates between different syntactic relations). See the paper for mor details.

epochs The number of epochs for which the model was trained.

cite The bibtex entry for the paper presenting the resource, that should be referenced in subsequent work building on or using the resource. It should be bibtex rather than biblatex, as most NLP publishers have not made the switch yet.

vocabulary The vocabulary metadata as described in *The vocab metadata*, which also includes the corpus metadata.

```
{
"class": "embeddings",
    "model": "",
    "window": ,
    "dimensionality": ,
    "context": "",
    "epochs": ,
    "cite": "",
    "vocabulary":    {
                     }
    "lib_version": "",
    "system_info": "",
}
```

1.3.4 The datasets metadata

The task datasets should be accompanied by the following metadata:

task The task for which the dataset is applicable, such as **word_analogy** or **word_relatedness**.

language A list containing the language codes for the corpus. There will be just one entry in case of monolingual corpora (e.g. _["en"]_), and for parallel or multilingual corpora there will be several (_["en", "de"]_).

name The (preferably short) name of the dataset, such as **WordSim353**.

description The freeform brief description of the dataset, preferably including anything special about this dataset that distinguishes it from other datasets for the same task.

domain The domain of the dataset, such as **news**, **encyclopedia**, **fiction**, **medical**, **spoken**, or **web**. We suggest using **general** only for datasets that do not target any particular domain.

date The date the resource was published.

source The source of the resource (e.g. a modification of another dataset, or something created by the authors from scratch or on the basis of some data that was not previously used for the same task).

project_page The URL of the page describing the dataset (if any)

version The version of the dataset (useful when you are developing one).

size The size of the dataset. The units depend on the task: it can be e.g. **353 pairs** for a similarity or analogy dataset.

cite The bibtex entry for the paper presenting the resource, that should be referenced in subsequent work building on or using the resource. It should be bibtex rather than biblatex, as most NLP publishers have not made the switch yet.

```
{
"class": "dataset",
    "task": "",
    "language": ["english"],
    "name": "",
    "description": "",
    "domain": "",
    "date": "",
    "source": "",
    "project_page": "",
    "version": "",
    "size": "",
    "cite": ""
}
```

1.3.5 The experiment metadata

As with the training of embeddings, different experiments involve different sets of metadata. The parameters of each model included in the Vecto library is described in the corresponding library page. In addition to that, the metadata for each experiment will automatically include the metadata for the dataset and embeddings (which also includes the corpus metadata).

Some of the generic metadata fields that are applicable to all experiments include:

name The (hopefully descriptive) name of the model, such as **LogisticRegression**.

task The type of the task that this model is applicable to (e.g. **word_analogy** or **text_classification**).

description A brief description of the implementation, preferably including its use case (e.g. a sample implementation in a some framework, a standard baseline for some task, a state-of-the-art model.)

author The author of the code (for unpublished models).

version The version of the implementation, if any.

date The date when the code was published or contributed.

source If the code is reimplementation of something else, this is the field to indicate it.

cite The bibtex entry for the paper presenting the code, that should be referenced in subsequent work building on or comparing with this implementation. It should be bibtex rather than biblatex, as most NLP publishers have not made the switch yet.

```
{
"class": "experiment",
    "name": "",
    "task": "",
    "description": "",
    "author": "",
    "version": "",
    "date": "",
    "source": "",
    "cite": ""
}
```

1.3.6 Accessing the metadata in Vecto

All metadata is accessible from vsmlib while you are experimenting on dozens of VSMs you have built, facilitating both parameter search for a particular task and observations on what properties of VSMs result in what aspects of their performance.

You can access the VSM metadata as follows:

The name of the model, which is the name directory in which it is stored. For models generated with VSMlib, interpretable folder names with parameters are generated automatically.

```
>>> print(my_vsm.name)
w2v_comb2_w8_n25_i6_d300_skip_300
```

You can also access the metadata as a Python dictionary:

```
>>> print(my_vsm.metadata)
{'size_dimensions': 300, 'dimensions': 300, 'size_window': '8'}
```

1.4 Where to get data?

This page lists some source corpora and pre-trained word vectors you can download.

1.4.1 Source corpora

English Wikipedia, August 2013 dump, pre-processed

- One-sentence per line, cleaned from punctuation
- One-word-per-line, parser tokenization (this is the version used in the non-dependency-parsed embeddings downloadable below, so use this one if you would like to have directly comparable embeddings)
- Dependency-parsed version (CoreNLP Stanford parser)

1.4.2 Pre-trained VSMs

English

Wikipedia vectors (dump of August 2013)

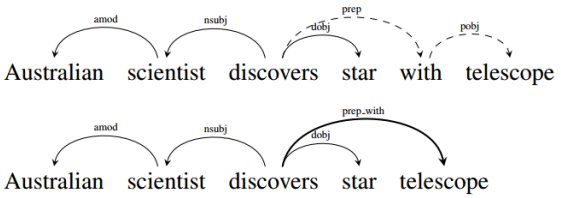
Here you can download 500-dimensional pre-trained vectors for the popular CBOW, Skip-Gram and GloVe VSMs - each in 4 kinds of context:

These embeddings were generated for the following paper. Please cite it if you use them in your research:

```
@inproceedings{LiLiuEtAl_2017_Investigating_Different_Syntactic_Context_Types_and_
↪Context_Representations_for_Learning_Word_Embeddings,
 title = {Investigating {{Different Syntactic Context Types}} and {{Context␣
↪Representations}} for {{Learning Word Embeddings}}},
 url = {http://www.aclweb.org/anthology/D17-1256},
 booktitle = {Proceedings of the 2017 {{Conference}} on {{Empirical Methods}} in {
↪{Natural Language Processing}}},
 author = {Li, Bofang and Liu, Tao and Zhao, Zhe and Tang, Buzhou and Drozd,␣
↪Aleksandr and Rogers, Anna and Du, Xiaoyong},
 year = {2017},
 pages = {2411--2421}}
```

You can also download the source corpus (one-word-per-line format) with which you can train other VSMs for fair comparison.

Each of the 3 models (CBOW, GloVe and Skip-Gram) is available in 5 sizes (25, 50, 100, 250, and 500 dimensions) and in 4 types of context: the traditional word linear context (which is used the most often), the dependency-based structured context, and also less common structured linear and word dependency context.



Context Representation \ Context Type	Linear	DEPS
unbound	australian, scientist, star, with	scientist, star, telescope
bound	australian/-2, scientist/-1, star/+1, with/+2	scientist/nsubj, star/dobj, telescope/prep_with

Unbound linear context (aka *word linear context*)

500 dimensions: word_linear_cbow_500d, word_linear_sg_500d, word_linear_glove_500d

250 dimensions: word_linear_cbow_250d, word_linear_sg_250d, word_linear_glove_250d

100 dimensions: word_linear_cbow_100d, word_linear_sg_100d, word_linear_glove_100d

50 dimensions: word_linear_cbow_50d, word_linear_sg_50d, word_linear_glove_50d

25 dimensions: word_linear_cbow_25d, word_linear_sg_25d, word_linear_glove_25d

Unbound dependency context (aka *word dependency context*)

500 dimensions: word_deps_CBOW_500d, word_deps_sg_500d, word_deps_glove_500d

250 dimensions: word_deps_cbow_250d, word_deps_sg_250d, word_deps_glove_250d

100 dimensions: word_deps_cbow_100d, word_deps_sg_100d, word_deps_glove_100d

50 dimensions: word_deps_cbow_50d, word_deps_sg_50d, word_deps_glove_50d

25 dimensions: word_deps_cbow_25d, word_deps_sg_25d, word_deps_glove_25d

Bound linear context (aka *structured linear context*)

500 dimensions: structured_linear_cbow_500d, structured_linear_sg_500d, structured_linear_glove_500d

250 dimensions: structured_linear_cbow_250d, structured_linear_sg_250d, structured_linear_glove_250d

100 dimensions: structured_linear_cbow_100d, structured_linear_sg_100d, structured_linear_glove_100d

50 dimensions: structured_linear_cbow_50d, structured_linear_sg_50d, structured_linear_glove_50d

25 dimensions: structured_linear_cbow_25d, structured_linear_sg_25d, structured_linear_glove_25d

Bound dependency context (aka *structured dependency context*)

500 dimensions: structured_deps_cbow_500d, structured_deps_sg_500d, structured_deps_glove_500d

250 dimensions: structured_deps_cbow_250d, structured_deps_sg_250d, structured_deps_glove_250d

100 dimensions: structured_deps_cbow_100d, structured_deps_sg_100d, structured_deps_glove_100d

50 dimensions: structured_deps_cbow_50d, structured_deps_sg_50d, structured_deps_glove_50d

25 dimensions: structured_deps_cbow_25d, structured_deps_sg_25d, structured_deps_glove_25d

The training parameters are as follows: window 2, negative sampling size is set to 5 for SG and 2 for CBOW. Distribution smoothing is set to 0.75. No dynamic context or "dirty" sub-sampling. The number of iterations is set to 2, 5 and 30 for SG, CBOW and GloVe respectively.

SVD vectors:

BNC, 100M words window 2, 500 dims, PMI; SVD C=0.6, 318 Mb, mirror

Russian

Araneum+Wiki+Proza.ru, 6B words window 2, 500 dims, PMI; SVD C=0.6, 2.3 Gb, mirror, paper to cite

```
@inproceedings{7396482,
author={A. Drozd and A. Gladkova and S. Matsuoka},
booktitle={2015 IEEE International Conference on Data Science and Data Intensive␣
↪Systems},
title={Discovering Aspectual Classes of Russian Verbs in Untagged Large Corpora},
year={2015},
pages={61-68},
doi={10.1109/DSDIS.2015.30},
month={Dec}}
```

1.5 Training new models

This page describes how to train vectors with the models that are currently implemented in VSMlib.

1.5.1 Word2vec

Word2vec is arguably the most popular word embedding model. We provide implementation of extended word2vec model, which can be trained on linear and dependency-based contexts, with bound and unbound context representations.

Additionally we provide an implementation which considers characters rather than words to be the minimal units. This enables it to take advantage of morphological information: as far as a word-level models such as word2vec is concerned, "walk" and "walking" are completely unrelated, except through similarities in their distributions.

To train word2vec embeddings vsmlib can be envoked via the command line interface:

```
>>> python3 -m vsmlib.embeddings.train_word2vec
```

The command line parameters are as

--dimensions size of embeddings

--context_type context type [linear' or 'deps'], for deps context, the annotated corpus is required

--context_representation context representation ['bound' or 'unbound']

--window window size')

--model base model type ['skipgram' or 'cbow']

--negative-size number of negative samples

--out_type output model type ["hsm": hierarchical softmax, "ns": negative sampling, "original": no approximation]

--subword specify if subword-level approach should be used ["none", "rnn"]

--batchsize	learning minibatch size
--gpu	GPU ID (negative value indicates CPU)
--epochs	number of epochs to learn
--maxWordLength	max word length (only used for char-level subword)
--path_vocab	path to the vocabulary
--path_corpus	path to the corpus
--path_out	path to save embeddings
--test	run in test mode
--verbose	verbose mode

Alternatively, word2vec training can be done though vsmlib python API.

```
>>> vsmlib.embeddings.train_word2vec.train(args)
```

The arguments are argparse.namespace identical to command line arguments. Instance of ModelDense is returned.

Realted papers: original w2v, Bofang, Mnih, subword.

```
@inproceedings{MikolovChenEtAl_2013_Efficient_estimation_of_word_representations_in_
↪vector_space,
 title = {Efficient Estimation of Word Representations in Vector Space},
 urldate = {2015-12-03},
 booktitle = {Proceedings of International Conference on Learning Representations␣
↪(ICLR)},
 author = {Mikolov, Tomas and Chen, Kai and Corrado, Greg and Dean, Jeffrey},
 year = {2013}}
```

```
@inproceedings{Li2017InvestigatingDS,
 title={Investigating Different Syntactic Context Types and Context Representations␣
↪for Learning Word Embeddings},
 author={Bofang Li and Tao Liu and Zhe Zhao and Buzhou Tang and Aleksandr Drozd and␣
↪Anna Rogers and Xiaoyong Du},
 booktitle={EMNLP},
 year={2017}}
```

1.6 Basic operations

1.6.1 Supported VSM formats

At the moment the following data formats are supported:

- .bin format of word2vec (the file has to be called “vectors.bin”)
- .npy arrays with separate vocab files
- .txt plain-text vectors
- sparse vectors in hp5 format

  todo fasttext .vec format?

1.6.2 Importing vectors

Vecto assumes a one-folder-per-vsm folder structure. All files related to the same vsm - the metadata, vectors, vocab files, etc. - must all be stored in one directory. If the vector files has the correct extension (.npy, .txt, .bin, .hp5), the library will attempt to "guess" the correct module to load it with.

```
>>> import vecto
>>> path_to_vsm = "/path/to/your/model"
>>> my_vsm = vecto.model.load_from_dir(path_to_vsm)
```

The name of the model is the name directory in which the vector files are stored. For models generated with Vecto, interpretable folder names with parameters are generated automatically.

```
>>> print(my_vsm.name)
w2v_comb2_w8_n25_i6_d300_skip_300
```

You can access the VSM metadata (recorded in metadata.json file located in the same directory as the VSM) as a Python dictionary:

```
>>> print(my_vsm.metadata)
{'size_dimensions': 300, 'dimensions': 300, 'size_window': '8'}
```

1.6.3 Getting top similar neighbors of a word

```
>>> my_vsm.get_most_similar_words("apple", cnt=5)
[['apple', 1.0000000999898755],
 ['fruit', 0.61400752577032369],
 ['banana', 0.58657183882050712],
 ['plum', 0.5850951585421692],
 ['apples', 0.58464719369713347]]
```

This method takes an optional `cnt` argument specifying how many top similar neighbors to output (the default is 10). Note that the top similar vector is always the target word itself.

If you need to compute nearest neighbors for many words, this function works faster if the VSM is normalized. If it was generated with vecto, the normalization will be recorded in metadata, and can be checked with `.normalized`. Vecto will automatically check for normalization and use the faster routine if possible. If not, you can first normalize your VSM with `.normalized()` method.

If for whatever reasons you need your VSM to not be normalized, you can use `.cache_normalized_copy()` method to cache normalized copy of embeddings. Please note that latter will consume additional memory.

.get_most_similar_vectors() enables you to do the same as `.get_most_similar_words()`, but searching the top neighbors by the vector representation rather than its label.

1.6.4 Words to vectors and back

First, you need to import your model from a directory that holds only that model (.npy, .bin, .hp5 or .txt formats) and any associated files.

getting the vector representation of a word

```
>>> my_vsm.get_row("apple")
array([-0.17980662,  0.27027196, -0.33250481,  ... -0.22577444], dtype=float32)
```

You can use the above top-similar function to get the label of the vector most corresponding to your vector in your VSM vocabulary:

```
>>> vsm.get_most_similar_vectors(vsm.get_row("apple"))
```

1.6.5 Filtering the vocabulary of a VSM

In certain cases it may be useful to filter the vocabulary of a pre-trained VSM, e.g. to ensure that two models you are comparing have the same vocabulary. Vecto provides a `.filter_by_vocab()` method that returns a new model instance, the vocabulary of which contains only the words in the provided Python list of words. The list can be empty.

```
>>> my_vsm.get_most_similar_words("cat", cnt=5)
[['cat', 1.0],
 ['monkey', 0.95726192],
 ['dog', 0.95372206],
 ['koala', 0.94773519],
 ['puppy', 0.94360757]]
>>> my_new_vsm = my_vsm.filter_by_vocab(["dog", "hotdog", "zoo", "hammer", "cat"])
>>> my_new_vsm.get_most_similar_words("cat", cnt=5)
[['cat', 1.0],
 ['dog', 0.95372206],
 ['hotdog', 0.84262532],
 ['hammer', 0.80627602],
 ['zoo', 0.7463485]]
```

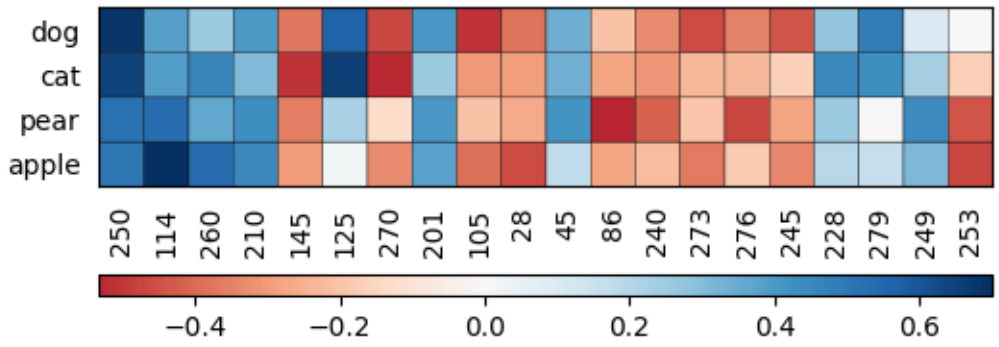
1.7 Visualization

When you have the numerical vectors for the units you are interested in, you can use all the goodies of matplotlib to create any kind of visualizaion you like. The visualize module of Vecto provides a few simple examples to get you started and/or quickly explore your model as you go.

The *visualize* module of vecto comes with several functions to quickly explore the representations.

1.7.1 Drawing features

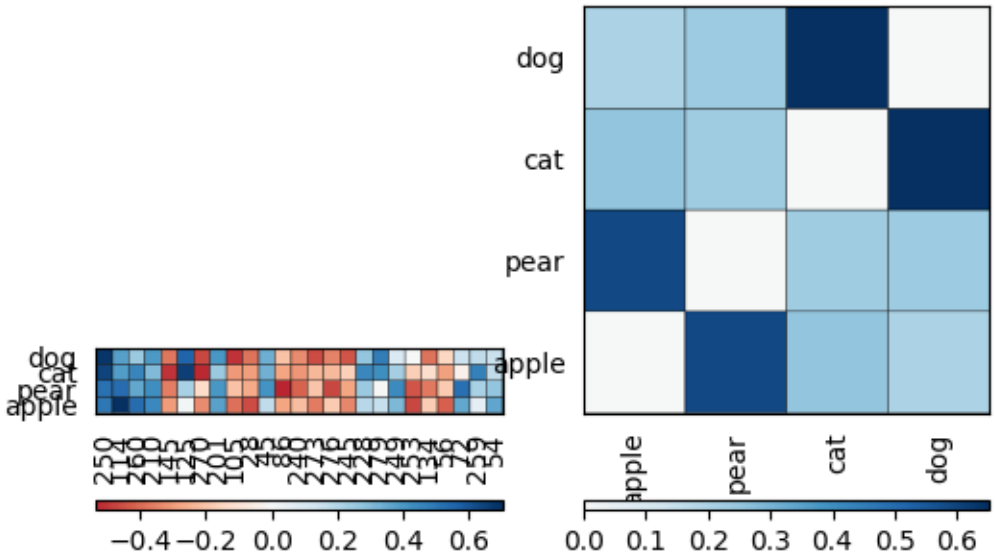
```
>>> from vecto import visualize as vz
>>> vs.draw_features(vsm, ["apple", "pear", "cat", "dog"], num_features=20)
```



TODO: how to interpret this.

1.7.2 Visualizing similarity between certain words.

```
>>> vs.draw_features_and_similarity(vsm, ["apple", "pear", "cat", "dog"])
```



The color intensity indicates the degre of similarity. We can see that apple is more similar to pear than to cat or dog, and the other way round.

1.7.3 Visualizing dimensions

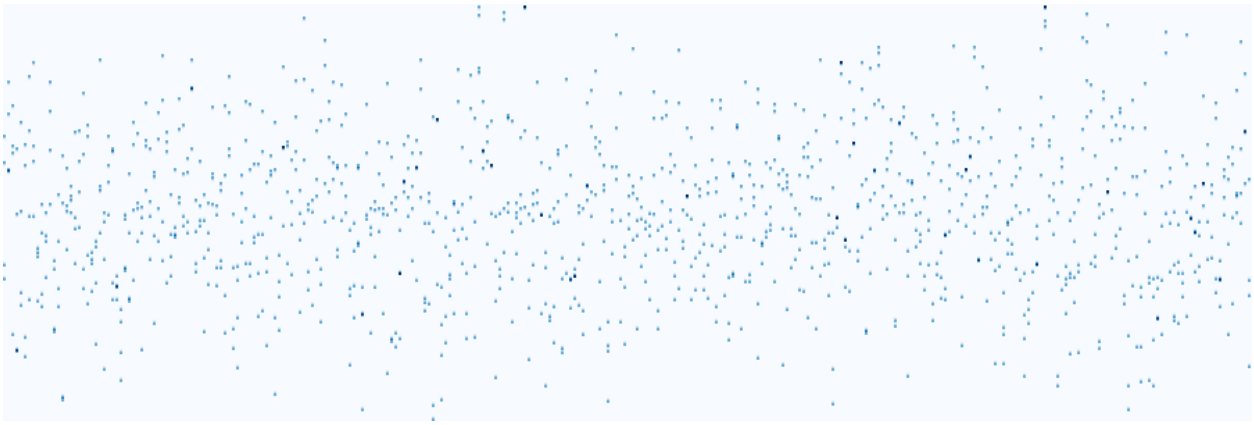
In a dense VSM, each dimension on its own is not likely to be an interpretable semantic feature on its own. Still, it is the overall pattern of the dimensios that encodes the meaning of any given language unit, and so it may be useful to visually inspect them.

```
>>> vs.std_to_img(vsm.get_row("apple"))
```

```
>>> vs.std_to_img(vsm.get_row("cat"))
```

The *rows_to_img* function displays only the end points of all dimensions in a given collection of vectors.

```
>>> vectors = vs.wordlist_to_rows(vsm, ["apple", "pear", "cat", "dog"])
>>> vs.rows_to_img_tips(vectors,max_y=0.8)
```



1.8 Intrinsic evaluation

1.8.1 Word analogy task

One of the de-facto standard intrinsic evaluations for word embeddings is the word analogy task. The dataset known as the Google test set became the de-facto standard for evaluating word embeddings, but it is not balanced and samples only 15 linguistic relations, with 19,544 questions in total. A newer dataset is BATS: it is considerably larger (98,000 questions) and is balanced: it contains 40 different relations of 4 types (inflections, derivational morphology, lexicographic and encyclopedic semantics) with 50 unique pairs per relation.

Vecto comes with the script to test 6 different methods of solving word analogies. You can run the script from command line, indicating the path to the config file as the only argument.

```
python3 -m vecto.benchmarks.analogy /path/to/config_analogy.yaml
```

The configuration file is structured as follows:

```
path_vectors: [
    "/path/to/your/vsm1/"
    "/path/to/your/vsm2/"
   ]

alpha: 0.6
# this is the exponent for Sigma values of SVD embeddings

normalize: true
# specifies if embeddings should be normalized

method: LRCos
# allowed values are 3CosAdd, 3CosAvg, 3CosMul, SimilarToB, SimilalarToAny,␣
↪PairDistance, LRCos and LRCosF

exclude: True
# specifies if question words should be excluded from possible answers

path_dataset: "/path/to/the/test/dataset"
```

(continues on next page)

(continued from previous page)

```
# path to dataset. last segment of the path will be interpreted as dataset name

path_results: "/path/where/to/save/results"
# Subfolders for datasets and embeddings willl be created automatically
```

Vecto also support direct call from **run(embeddings, options)** function. The **options** has the same parameters as that in **yaml** file. This function returns a dict, which indicate the word analogy results.

For example, the following lines can be used to get word analogy results:

```
path_model = "./test/data/embeddings/text/plain_no_file_header"
model = vecto.model.load_from_dir(path_model)
options = {}
options["path_dataset"] = "./test/data/benchmarks/analogy/"
options["path_results"] = "/tmp/vecto/analogy"
options["name_method"] = "3CosAdd"
vecto.benchmarks.analogy.analogy.run(model, options)
```

Dataset

The BATS dataset can be downloaded here. The script expects the input dataset to be a tab-separated file formatted as follows:

```
cat cats
apple apples
```

In many cases there is more than one correct answer; they are separated with slashes:

```
father  dad/daddy
flower  blossom/bloom
harbor  seaport/haven/harbour
```

There is a file with a word pairs list for each relation, and these files are grouped into folders by the type of the relation. You can also make your own test set to use in Vecto, formatted in the same way.

Analogy solving methods

Consider the analogy a:a' :: b:b' (a is to a' as b is to b'). The script implements 6 analogy solving methods:

Pair-based methods:

3CosAdd: $b' = argmax_{d \in V}(cos(b', b - a + a'))$, where $cos(u, v) = \frac{u \cdot v}{||u|| \cdot ||v||}$

PairDistance, aka PairDirection: $b' = argmax_{d \in V}(cos(b' - b, a' - a))$

3CosMul: $argmax_{b' \in V} \frac{cos(b', b) cos(b', a')}{cos(b', a) + \varepsilon}$ $\varepsilon = 0.001$ is used to prevent division by zero)

SimilarToB: returns the vector the most similar to the b.

SimilarToAny: returns the vector the most similar to any of a, a' and b vectors.

Set-based methods: (current state-of-the-art)

3CosAvg: $b' = argmax_{b' \in V}(cos(b', b + avg\_offset))$, where $avg\_offset = \frac{\sum_{i=0}^{m} a_i}{m} - \frac{\sum_{i=0}^{n} b_i}{n}$

LRCos $b' = argmax_{b' \in V}(P_{(b' \in target\_class)} * cos(b', b))$

LRCosF: a version of LRCos that attempts to only take into account the relevant distributional features.

Caveat: Analogy has been shown to be severely misinterpreted as evaluation task. First of all, all of the above methods are biased by distance in the distributional space: the closer the target is, the more likely you are to hit it. Therefore high scores on analogy task indicate basically to what extent the relations encoded by a given VSM match the relations in the dataset.

Therefore it would be better to not just provide an average score on the whole task, as it is normally done, but to look at the scores for different relations, as that may show what exactly the model is doing. Since everything cannot be close to everything, it is to be expected that success in one type of relations would come at the expense of others.

1.8.2 Correlation with human similarity/relatedness judgements

One of the first intrinsic evaluation metrics for distributional meaning representations was correlation with human judgements to what extent words are related. Roughly speaking, a good VSM should have tiger and zoo closer in the vector space than tiger and hammer, because tiger and zoo are intuitively more semantically related. There are several datasets with judgements of relatedness and similarity between pairs of words collected from human subjects. See (Turney 2006) for the distinction between relatedness and similarity (or relational and attributional similarity).

You can run this type of test in Vecto as follows:

```
>>> python3 -m vecto.benchmarks.similarity /path/to/config_similarity.yaml
```

The `config_similariy.yaml` file is structured as

```
path_vector: /path/to/your/vsm1/
path_dataset: /path/to/the/test/dataset
normalize: true      # specifies if embeddings should be normalized
```

Similar to word analogy task, Vecto also support direct call from **run(embeddings, options)** function. The following lines can be used to get word similarity results:

```
path_model = "./test/data/embeddings/text/plain_with_file_header"
model = vecto.model.load_from_dir(path_model)
options = {}
options["path_dataset"] = "./test/data/benchmarks/similarity/"
vecto.benchmarks.similarity.similarity.run(model, options)
```

The similarity/relatedness score file is assumed to have the following tab-separated format:

```
tiger   cat 7.35
book    paper   7.46
computer    keyboard    7.62
```

You can use any of the many available datasets, including:

- WordSim 353 (there is also a version of WordSim353 split into relatedness and similarity subsets)
- MEN
- SimLex
- Rare Words
- Radinsky Mturk data

Please refer to the pages of individual datasets for details on how they were collected and references to them. The collection of the above datasets in the same format can also be downloaded here.

Caveat: while similarity and relatedness tasks remain one of the most popular methods of evaluating word embeddings, they have serious methodological problems. Perhaps the biggest one is the unreliability of middle judgements:

while humans are good at distinguishing clearly related and clearly unrelated word pairs (e.g. *cat:tiger* vs *cat:malt*), there is no clear reason for rating any of the many semantic relations higher than the other (e.g. which is more related - *cat:tiger* or *cat:whiskers*)? It is thus likely that the human similarity scores reflect some psychological measures like speed of association and prototypicality rather than something purely semantic, and thus a high score on a similarity task should be interpreted accordingly. This would also explain why a high score on similarity or relatedness does not necessarily predict good performance on downstream tasks.

1.9 Extrinsic evaluation

The following tasks will soon be available via vecto:

- POS tagging
- Named entity recognition
- Chunking

1.10 Project roadmap

Vecto is work in progress. Everything that works at the moment is described in the present tutorial; feel free to get in touch if anything is not clear. Also, new functionality is coming in the nearest months, so check back for more features!

DONE	IN PROGRESS
General:	
• Loading various vsm formats: plain text, npy, binary, h5p • Metadata generation • Basic vector operations, efficient similarity search • VSM visualization	• Pretty data downloader for benchmarks
VSM generation:	
• word2vec • – Character-level VSM	• GloVe • SVD
VSM evaluation:	
• 6 methods of solving word analogies • similarity and relatedness tests • text classification • sequence labeling (POS-tagging, chunking, NER)	• natural language inference • language modeling • neural machine translation • subjectivity classification • and more!

CHAPTER 2

API reference

vecto is a library for all things related to vector space models in NLP

2.1 Submodules

embeddings	
vocabulary	
benchmarks	Collection of benchmarks and downstream tasks on embeddings

CHAPTER 3

Contribution Guide

This is a guide for all contributions to vecto. The development of vecto is happening on the official repository at GitHub.

3.1 Some quick notes:

Please send pull requests to the `dev` branch.

Pull requests must not lower test coverage score.

If you send a pull request, please make sure your code is pep8-compliant.

If you want to raise an ussue, please first do a quick search to see if it has already been reported. If so, it's often better to just leave a comment on an existing issue, rather than creating a new one.

Issues are for bug reports, feature requests etc. For usage-related questions please consult the tutorial; if something is not covered, raise an issue, and we will update the tutorial.

If there's an issue you would like to fix - this is very welcome, please get in touch.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

v

Index