Vec4IR – User's Guide

Lukas Galke

March 24, 2017

User's Guide

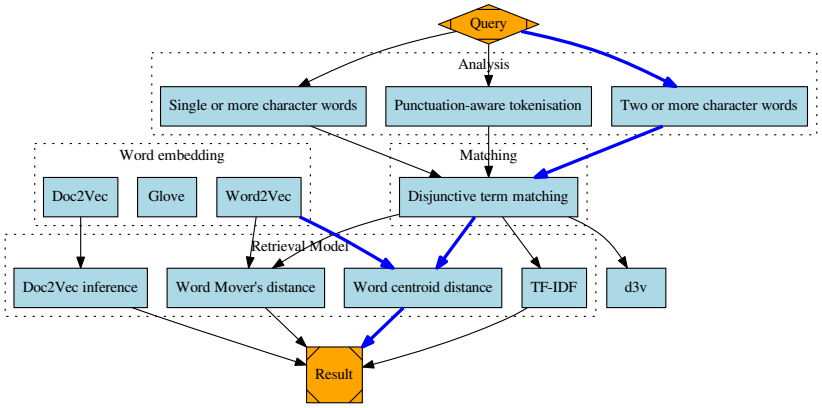

Figure 1: Detailed information retrieval pipeline as considered in the framework. The best-performing embedding-based configuration is highlighted with blue edges.

Philosophy

This information retrieval framework is not designed for production usage. Rather, the target is to *simulate* a practical information retrieval setting. In order to encourage research in this field, the focus lies on extensibility. Adding a new retrieval model for evaluation should be as easy as possible. The target audience are researchers evaluating their own retrieval models and curious data engineers, who want to find out which retrieval model fits their data best.

Definitions

In the following, We recapture the most important definitions that are relevant to the framework.

Information Retrieval (IR)

The information retrieval task can be defined as follows:

Given a corpus of documents D and a query q, return $\{d \in D \mid d \text{ relevant to } q\}$ in rank order.

A practical information retrieval system typically consists of at least the two components: matching, similarity scoring. Several other components can also be considered, such as query expansion, pseudo-relevance feedback, query parsing and the analysis process itself.

Matching

The matching operation refers to the initial filtering process of documents. In its easiest way the output of the matching operation contains all the document which contain *at least one* of the query terms. More sophisticated methods may also allow a fuzzy matching (up to some threshold in Levenshtein distance).

Similarity scoring

The scoring step refers to the process of assigning scores. The scores resemble the relevance of a specific document to the query. They are used to create a ranked ordering of the matched documents. This sorting happens either ascending, when considering distance scores, or descending in case of similarity scores.

Word embeddings and Word2Vec

A word embedding is a distributed (dense) vector representation for each word of a vocabulary.

Those word embeddings capture semantic and syntactic properties of the input texts. Interestingly, even arithmetic operations on the word vectors become meaningful: e.g. ‘$King$‘ − ‘$Queen$‘ = ‘Man‘ − ‘$Woman$‘. There are two popular approaches to learn a word embedding from raw text:

- Skip-Gram Negative Sampling or Word2Vec by Mikolov et al. (2013)
- Global Word Vectors or GloVe by Pennington, Socher, and Manning (2014)

Skip-gram negative sampling (or Word2Vec) is an algorithm based on a shallow neural network which aims to learn a word embedding. It is highly efficient, as it avoids dense matrix multiplication and does not require the full term co-occurrence matrix. Given some target word w_t, the intermediate goal is to train the neural network to predict the words in the c-neighborhood of w_t: $w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}$. First the word is directly associated to its respective vector, which as used as input for a (multinomial) logistic regression to predict the c-neighborhood words. The weights for the logistic regression are adjusted, as well as the vector itself (by back-propagation)., The Word2Vec algorithm employs negative sampling: additional k noise words which do **not** appear in the c-neighborhood are introduced as possible outputs, for which the desired output is known as `false`. Thus, the model does not reduce the weights to all other vocabulary words but only to those the sampled k noise words. If you then sample the noise words that they actually appear in a similar context as w_t, the model gets more and more fine-grained over the training epochs. In contrast to word to Word2Vec, the GloVe (Pennington, Socher, and Manning 2014) algorithm computes the whole term co-occurrence matrix for a given corpus. To obtain word vectors, the term co-occurrence matrix is factorized. The goal is that the euclidean dot product of each two word vectors match the log-probability of their words' co-occurrence.

Word centroid similarity (WCS)

An intuitive approach to employ word embeddings in information retrieval is the word centroid similarity (WCS). The representation for each document is the centroid of its respective word vectors. Since word vectors carry semantic information of the words, it is assumed that the centroid carries a notion of similarity. At query time, the centroid of the query's word vectors is computed and the cosine similarity to the centroids of the (matching) documents is used as a measure of relevance. Finally, the (matching) documents are sorted by descending cosine similarity of the centroids. In case the word frequencies are re-weighted according to inverse-document frequency (i.e. frequent words in the corpus are discounted), the technique is labelled IDF re-weighted word centroid similarity (IWCS).

Features

The key features of this information retrieval framework are:

- Simulation of a practical IR setting
- Native word embedding support
- Built-in evaluation in terms of `MAP`, `MRR`, `NDCG`, `precision`, `recall`, `f1`
- API design of `sklearn` (Buitinck et al. 2013) .
- Easy to extend by new retrieval models

Dependencies

Besides `python3` itself, the package `vec4ir` depends on the following python packages.

- numpy
- scipy
- gensim
- pandas
- networkx
- rdflib
- nltk
- sklearn
- pyyaml
- pyemd

Installation

As `vec4ir` is packaged as a python package, it can be installed by (if python `setuptools` are installed): We recommend, to install `numpy` and `scipy` in beforehand (either manually, or as binary system packages).

```
cd python-vec4ir; python3 setup.py install
```

In case anything went wrong with the installation of dependencies, try to install them manually:

```
pip3 install -r requirements.txt
```

We also provide a helper script to setup a new virtual environment. It can be invoked `setup.sh <venv-name>` to setup a new virtual environment. The newly created virtual environment has to be activated via `source <venv-name>` before performing the actual installation steps.

The evaluation script

The package includes a native command line script `ir_eval.py` for evaluation of an information retrieval pipeline. The pipeline of query expansion, matching and scoring is applied to a set of queries and the metrics MAP, MRR, NDCG, precision and recall are computed. Hence, the script may be used *as-is* for evaluation of your datasets or as a reference implementation for the usage of the framework. The behaviour of the evaluation script and the resulting information retrieval pipeline can be controlled by the following command-line arguments:

Meta options

- `-c, --config` Path to a configuration file, in which the file paths for the datasets and embedding models are specified.

General options

- `-d, --dataset` The data set to operate on. (mandatory)
- `-e, --embedding` The word embedding model to use. (mandatory)
- `-r, --retrieval-model` The retrieval model for similarity scoring. One of `tfidf` (default), `wcd`, `wmd`, `d2v`.
- `-q, --query-expansion` The expansion technique to apply on the queries. One of `wcd`, `eqe1`, `eqe2`.

Embedding options

- `-n, --normalize` Normalize the embedding model's word vectors.
- `-a, --all-but-the-top` All-but-the-top embedding post-processing (Mu, Bhat, and Viswanath 2017)
- `-t, --train` Number of epochs used for up-training out-of-vocabulary words .

Retrieval options

- `-I, --no-idf` Do *not* use IDF re-weighted word frequencies for aggregation of word vectors.
- `-w, --wmd` Fraction of *additional* documents to take into account for `wmd` retrieval model.

Output options

- `-o, --output` File path for writing the output.
- `-v, --verbose` Verbosity level.
- `-s, --stats` Compute out of vocabulary statistics and exit.

Evaluation options

- `-k` Number of documents to retrieve. Also relevant for the evaluation metrics.
- `-Q, --filter-queries` Drop queries, which contain out-of-vocabulary words.

- `-R, --replacement` Treatment for missing relevance information in the gold standard. Chose `drop` to disregard them (default) or `zero` to treat them as non-relevant.

Analysis (and therefore matching) options

- `-M, --no-matching` Do *not* conduct a matching operation.
- `-T, --tokenizer` The tokeniser for the matching operation. One of `sklearn`, `sword`, `nltk`.
- `-S, --dont-stop` Do *not* remove English stop-words.
- `-C, --cased` Conduct a case *sensitive* analysis.

Basic framework usage

We provide a minimal example including the matching operation and the TF-IDF retrieval model. As an example, assume we have two documents `"fox valley"` and `"dog nest"` and two queries `fox` and `dog`. First, we create an instance of the `Matching` class, whose optional arguments are passed directly to `sklearn`'s `CountVectorizer`.

```
>>> match_op = Matching()
```

In its default form, the Matching is a non-fuzzy binary `OR` matching. The `Matching` instance can be used after fitting a set of documents via `match_op.fit(documents)` via its `match_op.predict(query_string)` method to return the indices of the matching documents. However the preferred way to apply the matching operation is inside of a `Retrieval` instance. The `Retrieval` instance expects a retrieval model as mandatory argument, besides an optional `Matching` instance (and an optional query expansion instance). For now, we create an instance of the `Tfidf` class and pass it to the `Retrieval` constructor.

```
>>> tfidf = Tfidf()
>>> retrieval = Retrieval(retrieval_model=tfidf, matching=match_op)
>>> retrieval.fit(titles)
>>> retrieval.query("fox")
[0]
>>> retrieval.query("dog")
[1]
```

Under the hood, the `Retrieval` instance consults the `predict` method of the `matching` argument for a set of matching indices. The matching indices are passed as `indices` keyword argument to the `query` method of the retrieval model.

Employing word embeddings

Several supplied retrieval models make use of a word embedding. However those retrieval models do not learn a word embedding themselves, but take a `gensim` `Word2Vec` object as argument.

```
from gensim.models import Word2Vec
model = Word2Vec(documents['full-text'], min_count=1)
wcd = WordCentroidDistance(model)
RM = Retrieval(wcd, matching=match_op).fit(documents['full-text'])
```

Several embedding-based retrieval models are provided natively:

- **Word Centroid Distance** The cosine similarity between centroid of the document's word vectors and the centroid of the query's word vectors (`WordCentroidDistance(idf=False)`).
- **Word Centroid Distance** The cosine similarity between the document centroid and the query centroid after re-weighting the terms by inverse document frequency (`WordCentroidDistance(idf=True)`).
- **Word Mover's Distance** Optimizes the cost of moving from the words of the query to the words of the documents (`WordMoversDistance(complete=1.0)`). The optional `complete` parameter (between 0 and 1) allows to reduce the amount of documents considered relative to WCD. Setting `complete=0` results in re-ranking the documents returned by `WordCentroidDistance(idf=True)` with respect to Word Mover's Distance, while setting `complete=1.0` computes the Word Movers distance for all matched documents.
- **Doc2Vec Inference** The `Doc2VecInference` class expects a `gensim` `Doc2Vec` object as base model instead of a `Word2Vec` object. The model is used to infer document vectors for the documents and the queries. The cosine distance of inferred document vectors is used as a notion of similarity.

All these provided embedding-based retrieval models are designed for usage inside the `Retrieval` wrapper, i. e., they do not perform any matching operation by themselves.

Evaluating the model

To evaluate your retrieval model `RM` just invoke its `evaluate(X, Y)` method with `X` being a list of (`query_id`, `query_string`) pairs. The gold standard `Y` is a two-level data structure. The first level corresponds to the `query_id` and the second level to the `document_id`. Thus `Y[query_id][document_id]` should return the relevance number for the specific query-document pair. The exact

type of `Y` is flexible, it can be a list of dictionaries, a 2-dimensional `numpy` array, or a `pandas.DataFrame` with a (hierarchical) multi-index.

```
scores = RM.evaluate(X, Y)
```

The `evaluate(X, Y, k=None)` method returns a dictionary of various computed metrics per query. The scores can be manually reduced to their mean afterwards with a dictionary comprehension:

```
import numpy as np
mean_scores = {k : np.mean(v) for k,v in scores.items()}
```

The metrics (and therefore keys of the resulting scores dictionary consist of:

- `MAP` : Mean Average Precision (`@k`)
- `precision` : Precision (`@k`)
- `recall` : Recall (`@k`)
- `f1_score` : Harmonic mean of precision and recall
- `ndcg` : Normalized discounted cumulative gain (produces sensible scores with respect to the gold standard, even if k is given)
- `MRR` : Mean reciprocal rank (`@k`).

Extending by new models

According to the philosophy, implementing new retrieval model should be as easy as possible. In order to create a new retrieval model, one has to implement a class with at least 2 methods: `fit(X)` and `query(query, k=None, indices=None)`. In the following, we will demonstrate the implementation of a dummy retrieval model: the `GoldenRetriever` model which shuffles the results of the matching operation and returns them.

```
import random as RNG

class GoldenRetriever(RetriEvalMixin):
    def __init__(self, seed='bone'):
        RNG.seed(seed)


    def fit(self, documents):
        # Keep track of the documents
        self._fit_X = np.asarray(documents)


    def query(self, query, k=None, indices=None):
        # Pre-selection of matched documents
        if indices is not None:
            docs = self._fit_X[indices]
```

```
        else:
            docs = self._fit_X

        # Now we could do more magic with docs, or…

        ret = RNG.sample(range(docs.shape[0]), k)

        # Note that our result is assumed to be
        # relative to the matched indices
        # and NOT a subset of them

        return ret
```

The newly developed class contains three methods:

- **The constructor** All configuration is performed in the constructor, such that the model is ready to fit documents. No expensive computation is performed in the constructor.
- `fit(self, documents)` This method is called at index time, the retrieval model becomes aware of the documents and saves its internal representation of the documents.
- `query(self, query, k=None, indices=None)` The query method is called at query time. Beside the query string itself, it is expected to allow two keyword arguments: `k` resembling the number of desired documents, and `indices` which provides the indices of documents, that matched the query.

The reduction of the models own view on the documents (`docs = self._fit_X[indices]`) is important, since the returned indices are expected to be relative to this reduction (more details in the next section). These relative indices provide one key benefit. Oftentimes, the documents identifiers are not plain indices but string values. Using relative (to the matching) indices allows our retrieval model to disregard the fact that the document identifiers could be a string value or some other index, which does not match the position in our array of documents `X`. The presumably surrounding `Retrieval` class will keep track of the document identifiers for you and transpose the query's result `ret` to the identifier space.

The inheritance from `RetriEvalMixin` provides the `evaluation` method described above, which internally invokes the `query` method of the new retrieval model.

Matching, scoring, and query expansion

We provide a `Retrieval` class that implements the desired retrieval process of Figure 1. The `Retrieval` class consists of up to 3 components. It combines

the retrieval model (of the last section) as mandatory object and two optional objects: a matching operation and a query expansion object. Upon invocation of `fit(X)` the `Retrieval` class delegates the documents `X` to all prevalent components, i.e. it calls `fit(X)` on the matching object, the query expansion object, and the retrieval model object. A query expansion class is expected to provide a `fit` method which is called with the documents, along with a `transform` method which is called on the query. We provide more details on the implementation of a full information retrieval pipeline in the Developer's Guide.

Combining multiple fields and models

The `vec4ir` package also provides an experimental operator overloading API for combining multiple retrieval models.

```
RM_title = WordCentroidDistance().fit(documents['title'])
RM_content = Tfidf().fit(documents['full-text'])
RM = RM_title & RM_content  # fuzzy and: x+y - x*y
R = Retrieval(retrieval_model=RM)
```

On invocation of the `query` method on the combined retrieval model `RM`, both the model for the title and the model for the content get consulted and their respective scores are merged according to the operator. Operator overloading is provided for addition, multiplication and the binary `&` operator which implements `FUZZY-AND` $x \& y = x + y - x \cdot y$. For these `Combined` retrieval models, the consulted operand retrieval models are expected to return (`doc_id`, `score`) pairs in their result set. However, in this case the result set does not have to be sorted. Thus, the query method of the operand retrieval models is invoked with `sorted=False`. Still, the combined retrieval model `RM` keeps track of its nesting, such that the outer-most `Combined` instance *will* return a sorted list of results.

Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, et al. 2013. "API Design for Machine Learning Software: Experiences from the Scikit-Learn Project." *CoRR* abs/1309.0238. http://arxiv.org/abs/1309.0238.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. "Distributed Representations of Words and Phrases and Their Compositionality." In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a Meeting Held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, edited by Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, 3111–9. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.

Mu, Jiaqi, Suma Bhat, and Pramod Viswanath. 2017. "All-but-the-Top: Simple and Effective Postprocessing for Word Representations." *CoRR* abs/1702.01417.

http://arxiv.org/abs/1702.01417.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. "Glove: Global Vectors for Word Representation." In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A Meeting of Sigdat, a Special Interest Group of the ACL*, edited by Alessandro Moschitti, Bo Pang, and Walter Daelemans, 1532–43. ACL. http://aclweb.org/anthology/D/D14/D14-1162.pdf.