Magnitude: A Fast, Efficient Universal Vector Embedding Utility Package

Ajay Patel
Plasticity Inc.
San Francisco, CA
ajay@plasticity.ai

Alexander Sands
Plasticity Inc.
San Francisco, CA
alex@plasticity.ai

Chris Callison-Burch
Computer and Information
Science Department
University of Pennsylvania
ccb@upenn.edu

Marianna Apidianaki
LIMSI, CNRS
Université Paris-Saclay
91403 Orsay, France
marapi@seas.upenn.edu

Abstract

Vector space embedding models like word2vec, GloVe, and fastText are extremely popular representations in natural language processing (NLP) applications. We present Magnitude, a fast, lightweight tool for utilizing and processing embeddings. Magnitude is an open source Python package with a compact vector storage file format that allows for efficient manipulation of huge numbers of embeddings. Magnitude performs common operations up to 60 to 6,000 times faster than Gensim. Magnitude introduces several novel features for improved robustness like out-of-vocabulary lookups.

1 Introduction

Magnitude is an open source Python package developed by Ajay Patel and Alexander Sands (Patel and Sands, 2018). It provides a full set of features and a new vector storage file format that make it possible to use vector embeddings in a fast, efficient, and simple manner. It is intended to be a simpler and faster alternative to current utilities for word vectors like Gensim (Řehůřek and Sojka, 2010).

Magnitude's file format (".magnitude") is an efficient universal vector embedding format. The Magnitude library implements on-demand lazy loading for faster file loading, caching for better performance of repeated queries, and fast processing of bulk key queries. Table 1 gives speed benchmark comparisons between Magnitude and Gensim for various operations on the Google News pre-trained word2vec model (Mikolov et al., 2013). Loading the binary files containing the word vectors takes Gensim 70 seconds, versus 0.72 seconds to load the corresponding Magnitude file, a 97x speed-up. Gensim uses 5GB of RAM versus 18KB for Magnitude.

Metric	Cold	Warm
Initial load time	97x	–
Single key query	1x	110x
Multiple key query (n=25)	68x	3x
k-NN search query (k=10)	1x	5,935x

Table 1: Speed comparison of Magnitude versus Gensim for common operations. The 'cold' column represents the first time the operation is called. The 'warm' column indicates a subsequent call with the same keys.

Magnitude implements functions for looking up vector representations for misspelled or out-of-vocabulary words, quantization of vector models, exact and approximate similarity search, concatenating multiple vector models together, and manipulating models that are larger than a computer's main memory. Magnitude's ease of use and simple interface combined with its speed, efficiency, and novel features make it an excellent tool for cases ranging from applications used in production environments to academic research to students in natural language processing courses.

2 Motivation

Magnitude offers solutions to a number of problems with current utilities.

Speed: Existing utilities are prohibitively slow for iterative development. Many projects use Gensim to load the Google News word2vec model directly from a ".bin" or ".txt" file multiple times. It can take between a minute to a minute and a half to load the file.

Memory: A production web server will run multiple processes for serving requests. Running Gensim, in the same configuration, will consume >4GB of RAM usage per process.

Code duplication: Many developers duplicate effort by writing commonly used routines that are not provided in current utilities. Namely, routines for concatenating embeddings, bulk key lookup, out-of-vocabulary search, and building indexes for approximate k-nearest neighbors.

The Magnitude library uses several well-engineered libraries to achieve its performance improvements. It uses SQLite[1] as its underlying data store, and takes advantage of database indexes for fast key lookups and memory mapping. It uses NumPy[2] to achieve significant performance speedups over native Python code using computations that follow the Single Instruction, Multiple Data (SIMD) paradigm. It uses spatial indexes to perform fast exact similarity search and Annoy[3] to perform approximate k-nearest neighbors in the vector space. To perform feature hashing, it uses xxHash[4], an extremely fast non-cryptographic hash algorithm, working at speeds close to RAM limits. Magnitude's file format uses LZ4 compression[5] for compact storage.

3 Design Principles

Several design principles guided the development of the Magnitude library:

- The API should be intuitive and beginner friendly. It should have sensible defaults instead of requiring configuration choices by the user. The option to configure every setting should still be provided to power users.
- The out of the box configuration should be fast and memory efficient for iterative development. It should be suitable for deployment in a production environment. Using the same configuration in development and production reduces bugs and makes deployment easier.
- The library should use lazy loading whenever possible to remain fast, responsive, and memory efficient during development.
- The library should aggressively index, cache, and use memory maps to be fast, responsive, and memory efficient for production.
- The library should be able to process data that is too large to fit into a computer's main memory.
- The library should be thread-safe and employ memory mapping to reduce duplicated memory resources when multiprocessing.
- The interface should act as a generic key-vector store and remain agnostic to underlying models (like word2vec, GloVe, and fastText) and remain useable for other domains that use vector embeddings like computer vision (Babenko and Lempitsky, 2016).

Gensim offers several speed ups of its operations, but these are largely only accessible through advanced configuration. For example, by re-exporting a ".bin", ".txt", or ".vec" file into its own native format that can be memory-mapped. Magnitude makes this easier by providing a default configuration and file format that requires no extra configuration to make development and production workloads run efficiently out of the box.

4 Getting Started with Magnitude

The system consists of a Python 2.7 and Python 3.x compatible package (accessible through the PyPI index[6] or GitHub[7]) with utilities for using the ".magnitude" format and converting to it from other popular embedding formats.

4.1 Installation

Installation for Python 2.7 can be performed using the `pip` command:

```
pip install pymagnitude
```

Installation for Python 3.x can be performed using the `pip3` command:

```
pip3 install pymagnitude
```

4.2 Basic Usage

Here is how to construct the Magnitude object, query for vectors, and compare them:

[1] https://www.sqlite.org/
[2] http://www.numpy.org/
[3] https://github.com/spotify/annoy
[4] https://xxhash.org/
[5] http://www.lz4.org/
[6] https://pypi.org/project/pymagnitude/
[7] https://github.com/plasticityai/magnitude

```
from pymagnitude import *
vectors = Magnitude("./vecs.magnitude")
k = vectors.query("king")
q = vectors.query("queen")
vectors.similarity(k,q) # 0.6510958
```

Magnitude queries return almost instantly and are memory efficient. It uses lazy loading directly from disk, instead of having to load the entire model into memory. Additionally, Magnitude supports nearest neighbors operations, finding all words that are closer to a key than another key, and analogy solving (optionally with Levy and Goldberg (2014)'s 3CosMul function):

```
vectors.most_similar(k, topn=5)
#[('king', 1.0), ('kings', 0.71),
# ('queen', 0.65), ('monarch', 0.64),
# ('crown_prince', 0.62)]

vectors.most_similar(q, topn=5)
#[('queen', 1.0), ('queens', 0.74),
#('princess', 0.71), ('king', 0.65),
# ('monarch', 0.64)]

vectors.closer_than("queen", "king")
#['queens', 'princess']

vectors.most_similar(positive =
["woman", "king"], negative = ["man"])
# queen
vectors.most_similar_cosmul(positive =
["woman", "king"], negative = ["man"])
# queen
```

In addition to querying single words, Magnitude also makes it easy to query for multiple words in a single sentence and multiple sentences:

```
vectors.query("play")
# Returns: a vector for the word
vectors.query(["play", "music"])
# Returns: an array with two vectors
vectors.query([
 ["play", "music"],
 ["turn", "on", "the", "lights"],
]) # Returns: 2D array with vectors
```

4.3 Advanced Features

OOVs: Magnitude implements a novel method for handling out-of-vocabulary (OOV) words. OOVs frequently occur in real world data since pre-trained models are often missing slang, colloquialisms, new product names, or misspellings. For example, while *uber* exists in Google News word2vec, *uberx* and *uberxl* do not. These products were not available when Google News corpus was built. Strategies for representing these words include generating random unit-length vectors for each unknown word or mapping all unknown words to a token like "UNK" and representing them with the same vector. These solutions are not ideal as the embeddings will not capture semantic information about the actual word. Using Magnitude, these OOV words can be simply queried and will be positioned in the vector space close to other OOV words based on their string similarity:

```
"uber" in vectors # True
"uberx" in vectors # False
"uberxl" in vectors # False
vectors.query("uberx")
  # Returns: [ 0.0507, -0.0708, ... ]
vectors.query("uberxl")
  # Returns: [ 0.0473, -0.08237, ... ]
vectors.similarity("uberx", "uberxl")
  # Returns: 0.955
```

A consequence of generating OOV vectors is that misspellings and typos are also sensibly handled:

```
"missispi" in vectors # False
"discrimnatory" in vectors # False
"hiiiiiiiiii" in vectors # False
vectors.similarity(
  "missispi",
  "mississippi"
) # Returns: 0.359
vectors.similarity(
  "discrimnatory",
  "discriminatory"
) # Returns: 0.830
vectors.similarity(
  "hiiiiiiiiii",
  "hi"
) # Returns: 0.706
```

The OOV handling is detailed in Section 5.

Concatenation of Multiple Models: Magnitude makes it easy to concatenate multiple types of vector embeddings to create combined models.

```
w2v = Magnitude("GNews.300d.magnitude")
gv = Magnitude("glove.50d.magnitude")
vectors = Magnitude(w2v, gv) # concat
vectors.query("cat")
# Returns: 350d NumPy array
# 'cat' from w2v and 'cat' from gv
vectors.query(("cat", "cats"))
# Returns: 350d NumPy array
# 'cat' from w2v and 'cats' from gv
```

Adding Features for Part-of-Speech Tags and Syntax Dependencies to Vectors: Magnitude can directly turn a set of keys (like a POS tag set) into vectors. Given an approximate upper bound on the number of keys and a namespace, it uses the hashing trick (Weinberger et al., 2009) to create an appropriate length dimension for the keys.

```
pos_vecs = FeaturizerMagnitude(
  100, namespace = "POS")
```

Metric	Speed
Exact k-NN	0.9155s
Approx. k-NN (k=10, effort = 1.0)	0.1873s
Approx. k-NN (k=10, effort = 0.1)	0.0199s

Table 2: Approximate nearest neighbors significantly speeds up similarity searches compared to exact search. Reducing the amount of allowed effort further speeds the approximate k-NN search.

```
pos_vecs.dim # 4
# number of dims automatically
# determined by Magnitude from 100
pos_vecs.query("NN")
dep_vecs = FeaturizerMagnitude(
  100, namespace = "Dep")
dep_vecs.dim # 4
dep_vecs.query("nsubj")
```

This can be used with Magnitude's concatenation feature to combine the vectors for words with the vectors for POS tags or dependency tags. Homonyms show why this may be useful:

```
vectors = Magnitude(vecs, pos_vecs,
                    dep_vecs)
vectors.query([
    ("Buffalo", "JJ", "amod"),
    ("buffalo", "NNS", "nsubj"),
    ("Buffalo", "JJ", "amod"),
    ("buffalo", "NNS", "nsubj"),
    ("buffalo",  "VBP", "rcmod"),
    ("buffalo",  "VB", "ROOT"),
    ("Buffalo",  "JJ", "amod"),
    ("buffalo",  "NNS", "dobj")
  ]) # array of size 8 x (300 + 4 + 4)
```

Approximate k-NN We support approximate similarity search with the `most_similar_approx` function. This finds the approximate nearest neighbors more quickly than the exact nearest neighbors search performed by the `most_similar` function. The method accepts an `effort` argument which accepts the range $[0.0, 1.0]$. A lower `effort` will reduce accuracy, but increase speed. A higher `effort` does the reverse. This trade-off works by searching more- or less-indexed trees. Our approximate k-NN is powered by Annoy, an open source library released by Spotify. Table 2 compares the speed of various configurations for similarity search.

5 Details of OOV Handling

Facebook's fastText (Bojanowski et al., 2016) provides similar OOV functionality to Magnitude's. Magnitude allows for OOV lookups for any embedding model, including older models like word2vec and GloVe (Mikolov et al., 2013; Pennington et al., 2014), which did not provide OOV support. Magnitude's OOV method can be used with existing embeddings because it does not require any changes to be made at training time like fastText's method does.

Constructing vectors from character n-grams: We generate a vector for an OOV word w based on the character n-gram sequences in the word. First, we pad the word with a character at the beginning of the word and at the end of the word. Next, we generate the set of all character-ngrams in w (denoted with the fuction CGRAM_w) between length 3 and 6, following Bojanowski et al. (2016), although these parameters are tunable arguments in the Magnitude converter. We use the set of character n-grams C to construct a vector $\text{OOV}_d(w)$ with d dimensions to represent the word w. Each unique character n-gram c from the word contributes to the vector through a **p**seudo**r**andom **v**ector **g**enerator function PRVG. Finally, the vector is normalized.

$$C = \text{CGRAM}_w(3, 6)$$

$$\text{oov}_d(w) = \sum_{c \in C} \text{PRVG}_{\text{H}(c)}(-1.0, 1.0, d)$$

$$\text{OOV}_d(w) = \frac{\text{oov}_d(w)}{|\text{oov}_d(w)|}$$

PRVG's random number generator is seeded by the value "seed", which generates uniformly random vectors of dimension size d, with values in the range of -1 to 1. The hashing function H produces a 32 bit hash of its input using xxHash. $\text{H} : \{0,1\}^* \rightarrow \{0,1\}^{32}$. Since the PRVG's seed is only conditioned upon the word w, the output is deterministic across different machines.

This character n-gram-based method will generate highly similar vectors for a pair of OOVs with similar spellings, like *uberx* and *uberxl*. However, they will not be embedded close to similar in-vocabulary words like *uber*.

Interpolation with in-vocabulary words To handle matching OOVs to in-vocabulary words, we first define a function $\text{MATCH}_k(a, b, w)$. $\text{MATCH}_k(a, b, w)$ returns the normalized mean of the vectors of the top k most string-similar in-vocabulary words using the full-text SQLite index. In practice, we use the top 3 most string-similar words. These are then used to interpolate the values for the vector representing the OOV

word. 30% of the weight for each value comes from the pseudorandom vector generator based on the OOV's n-grams, and the remaining 70% comes from the values of the 3 most string similar in-vocabulary words:

$$\text{oov}_d(w) = [0.3 * \text{OOV}_d(w) + 0.7 * \text{MATCH}_3(3, 6, w)]$$

Morphology-aware matching For English, we have implemented a nuanced string similarity metric that is prefix- and suffix-aware. While *uberification* has a high string similarity to *verification* and has a lower string similarity to *uber*, good OOV vectors should weight stems more heavily than suffixes. Details of our morphology-aware matching are omitted for space.

Other matching nuances We employ other techniques when computing the string similarity metric, such as shrinking repeated character sequences of three or more to two (*hiiiiiiii* → *hii*), ranking strings of a similar length higher, and ranking strings that share the same first or last character higher for shorter words.

6 File Format

To provide efficiency at runtime, Magnitude uses a custom ".magnitude" file format instead of ".bin", ".txt", or ".vec" that word2vec, GloVe, and fastText use (Mikolov et al., 2013; Pennington et al., 2014; Joulin et al., 2016). The ".magnitude" file is a SQLite database file. There are 3 variants of the file format: Light, Medium, Heavy. Heavy models have the largest file size but support all of the Magnitude library's features. Medium models support all features except approximate similarity search. Light models do not support approximate similarity searches or interpolated OOV lookups, but they still support basic OOV lookups. See Figure 1 for more information about the structure and layout of the ".magnitude" format.

Converter The software includes a command-line converter utility for converting word2vec (".bin", ".txt"), GloVe (".txt"), or fastText (".vec") files to Magnitude files. They can be converted with the command:

```
python -m pymagnitude.converter
  -i "/path/to/vecs.(bin|txt|vec)"
  -o "/path/to/vecs.magnitude"
```

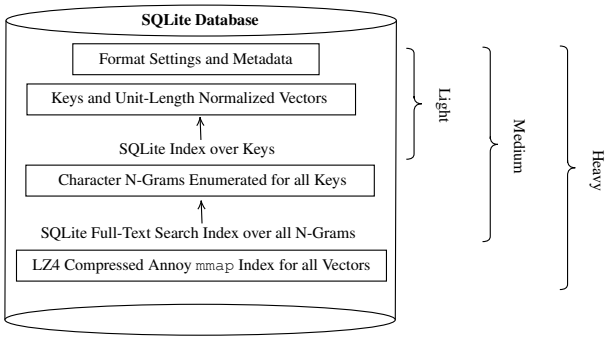

Figure 1: Structure of the ".magnitude" file format and its Light, Medium, and Heavy variants.

The input format will automatically be determined by the extension and the contents of the input file. When the vectors are converted, they will also be unit-length normalized. This conversion process only needs to be completed once per model. After converting, the Magnitude file format is static and it will not be modified or written to in order to make concurrent read access safe.

By default, the converter builds a Medium ".magnitude" file. Passing the `-s` flag will turn off encoding of subword information, and result in a Light flavored file. Passing the `-a` flag will turn on building the Annoy approximate similarity index, and result in a Heavy flavored file. Refer to the documentation[8] for more information about conversion configuration options.

Quantization The converter utility accepts a `-p <PRECISION>` flag to specify the decimal precision to retain. Since underlying values are stored as integers instead of floats, this is essentially quantization[9] for smaller model footprints. Lower decimal precision will create smaller files, because SQLite can store integers with either 1, 2, 3, 4, 6, or 8 bytes.[10] Regardless of the precision selected, the library will create `numpy.float32` vectors. The datatype can be changed by passing `dtype=numpy.float16` to the `Magnitude` constructor.

7 Conclusion

Magnitude is a new open source Python library and file format for vector embeddings. It makes it easy to integrate embeddings into applications

[8] https://github.com/plasticityai/magnitude#file-format-and-converter
[9] https://www.tensorflow.org/performance/quantization
[10] https://www.sqlite.org/datatype3.html

and provides a single interface and configuration that is suitable for both development and production workloads. The library and file format also enable novel features like OOV handling that allow models to be more robust to noisy data. The simple interface, ease of use, and speed of the library, compared to other utilities like Gensim, will enable use by beginners to NLP and individuals in educational environments, such as university NLP and AI courses.

Pre-trained word embeddings have been widely adopted in NLP. Researchers in computer vision have started using pre-trained vector embedding models like Deep1B (Babenko and Lempitsky, 2016) for images. The Magnitude library intends to stay agnostic to various domains, instead providing a generic key-vector store and interface that is useful for all domains and for research that crosses the boundaries between NLP and vision (Hewitt et al., 2018).

8 Software and Data

We release the Magnitude package under the permissive MIT open source license. The full source code and pre-converted ".magnitude" models are on GitHub. The full documentation for all classes, methods, and configurations of the library can be found at https://github.com/plasticityai/magnitude, along with example usage and tutorials.

We have pre-converted several popular embedding models (Google News word2vec, Stanford GloVe, and Facebook fastText) to ".magnitude" in all its variants (Light, Medium, and Heavy). You can download them from https://github.com/plasticityai/magnitude#pre-converted-magnitude-formats-of-popular-embeddings-models.

Acknowledgments

We would like to thank Erik Bernhardsson for the useful feedback on integrating Annoy indexing into Magnitude and thank the numerous contributors who have opened issues, reported bugs, or suggested technical enhancements for Magnitude on GitHub.

This material is funded in part by DARPA under grant number HR0011-15-C-0115 (the LORELEI program) and by NSF SBIR Award #IIP-1820240. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes. The views and conclusions contained in this publication are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA, the NSF, and the U.S. Government. This work has also been supported by the French National Research Agency under project ANR-16-CE33-0013.

References

Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, Las Vegas, NV.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching word vectors with subword information. *CoRR*, abs/1607.04606.

John Hewitt, Daphne Ippolito, Brendan Callahan, Reno Kriz, Derry Tanti Wijaya, and Chris Callison-Burch. 2018. Learning Translations via Images with a Massively Multilingual Image Dataset. In *Proceedings of ACL*, pages 2566–2576, Melbourne, Australia.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *CoRR*, abs/1607.01759.

Omer Levy and Yoav Goldberg. 2014. Linguistic regularities in sparse and explicit word representations. In *Proceedings of CoNLL*, pages 171–180, Ann Arbor, Michigan.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR*, abs/1301.3781.

Ajay Patel and Alex Sands. 2018. plasticityai/magnitude: Release 0.1.22. https://doi.org/10.5281/zenodo.1255637.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of EMNLP*, pages 1532–1543, Doha, Qatar.

Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature Hashing for Large Scale Multitask Learning. In *Proceedings of ICML*, pages 1113–1120, New York, NY.

A Benchmark Comparisons

All benchmarks[11] were performed on the Google News pre-trained word vectors, "GoogleNews-vectors-negative300.bin" (Mikolov et al., 2013) for Gensim and on the "GoogleNews-vectors-negative300.magnitude"[12] for Magnitude, with a MacBook Pro (Retina, 15-inch, Mid 2014) 2.2GHz quad-core Intel Core i7 @ 16GB RAM on a SSD over an average of trials where feasible. We are explicitly not using Gensim's memory-mapped native format as it requires extra configuration from the developer and is not provided out of the box from Gensim's data repository [13].

Metric	**Gensim (Řehůřek and Sojka, 2010)**	**Magnitude Light**	**Magnitude Medium**	**Magnitude Heavy**
Initial load time	70.26s	**0.7210s**	— [a]	— [a]
Cold single key query	**0.0001s**	**0.0001s**	— [a]	— [a]
Warm single key query *(same key as cold query)*	0.0044s	**0.00004s**	— [a]	— [a]
Cold multiple key query *(n=25)*	3.0050s	**0.0442s**	— [a]	— [a]
Warm multiple key query *(n=25) (same keys as cold query)*	**0.0001s**	**0.00004s**	— [a]	— [a]
First `most_similar` search query *(n=10) (worst case)*	**18.493s**	247.05s	— [a]	— [a]
First `most_similar` search query *(n=10) (average case) (w/ disk persistent cache)*	18.917s	**1.8217s**	— [a]	— [a]
Subsequent `most_similar` search *(n=10) (different key than first query)*	0.2546s	**0.2434s**	— [a]	— [a]
Warm subsequent `most_similar` search *(n=10) (same key as first query)*	0.2374s	**0.00004s**	**0.00004s**	**0.00004s**
First `most_similar_approx` search query *(n=10, effort=1.0) (worst case)*	N/A [b]	N/A	N/A	**29.610s**
First `most_similar_approx` search query *(n=10, effort=1.0) (average case) (w/ disk persistent cache)*	N/A	N/A	N/A	**0.9155s**
Subsequent `most_similar_approx` search *(n=10, effort=1.0) (different key than first query)*	N/A	N/A	N/A	**0.1873s**
Subsequent `most_similar_approx` search *(n=10, effort=0.1) (different key than first query)*	N/A	N/A	N/A	**0.0199s**
Warm subsequent `most_similar_approx` search *(n=10, effort=1.0) (same key as first query)*	N/A	N/A	N/A	**0.00004s**
File size	**3.64GB**	4.21GB	5.29GB	10.74GB
Process memory (RAM) utilization	4.875GB	**18KB**	— [a]	— [a]
Process memory (RAM) utilization after 100 key queries	4.875GB	**168KB**	— [a]	— [a]
Process memory (RAM) utilization after 100 key queries + similarity search	8.228GB [c]	**342KB** [d]	— [a]	— [a]

[a] Denotes the same value as the previous column.

[b] Gensim does support approximate similarity search, but not out of the box as the index must be built manually with `gensim.similarities.index` first which is a slow operation.

[c] Gensim has an option to not duplicate unit-normalized vectors in memory, but still requires up to 8GB of memory allocation while processing, before dropping down to half the memory. Moreover, this option is not on by default.

[d] Magnitude uses `mmap` to read from the disk, so the OS will still allocate pages of memory, when memory is available, in its file cache, but it can be shared between processes and is not managed within each process for extremely large files which is a performance win.

Table 3: Benchmark comparisons between Gensim, Magnitude Light, Magnitude Medium, and Magnitude Heavy.

[11] https://github.com/plasticityai/magnitude/blob/master/tests/benchmark.py

[12] http://magnitude.plasticity.ai/word2vec+approx/GoogleNews-vectors-negative300.magnitude

[13] https://github.com/RaRe-Technologies/gensim-data