Wordly: A Reverse Dictionary

Team Members: Jared Katzman & Tim Follo

GitHub: https://github.com/HombreDeGatos/Wordly

Abstract

What's that word?

It's what happens when you put a lot of pressure on something to shrink it?

It's on the tip of my tongue...

Wordly is a reverse dictionary decision system that could answer this question. More flexible and usable than a thesaurus, Wordly is a complete cognitive-based decision system (a thesaurus query, to the contrary, would be based on a large set of granular word congruence rules). It can take any novel, user-defined *description* of a word and produce the *word* the user is trying to describe.

It might have taken you, a human proficient in the english language and familiar with this mode of forgetfullness (maybe even for this exact situation!) a moment beofre guessing: "compress". You read all the words in the description, then formed some semantic representation from the input (this is the part humans naturally excel at, while machines struggle), then searched your memory for a matching word. But your memory is fickle! You can know exactly what to look for, but maybe it has been years since you thought of "compress" that way, and it simply doesnt show up while you rack your brain.

This simple, fun example demonstrates why a user might be interested in interacting with our system as presented here. As this document will show, we have built a user interface that is tailored to this use. However, the simplicity here obfuscates the complexity of such a decision, which boils down to choosing a single word that captures the meaning of a series of words. Wordly can map an infinitely large set of word *descriptions* to a finite set of *precise words*. We should not overlook the deeper potential of such a system: a ready-made method for assigning semantic meaning to word strings could be used in a number of language-processing contexts.

The rest of this paper will focus on the design and implementation details of this system, as well as results and analysis of performance. The rest of this section give a brief overview of the major concepts involved.

The first concept we borrow from computational linguistics and Natural Language Processing (NLP). An age

old problem in NLP is how to represent semantics, or meaning. One method is to use semantic vectors, where the meaning of a word is encoded in a high dimensional vector whose position relative to other words reflects their related meanings.

Under the hood, Wordly makes its decisions by using a series of weights that have been pre-computed by a recurrent nueral network (RNN) to choose the most important parts of the user input. The Network then produces a vectorized representation of the entire query string, or *description*.

The RNN 'decided' on the weights when it was 'trained' on a large set of dictionary *definitions*. The training process involves performing thousands of computations to minimize the euclidian distance between a predicted *word vector* and the *definition* entry's true value; thus, the RNN learns over time how to set the weights such that all definitions, on average, produce vectors that are close to their partner *words*.

When the user queries our reverse dictionary with a *description*, Wordly performs a mathematical operation on the vectorized version of the query: It is able to compare words to the semantic representation of the *description* and use this to test candidates. We decide to show the user the 5 closest words in vector space, in hopes that one of them is the word the user is thinking of.

This process, though a simplification, is modeled off of the way humans actually think about language, and as such this is an advanced cognitive decision model. Rather than simply matching preconceived sets of definitions, Wordly handles novel data and is only limited by the vocabulary we have trained it on and the quality of the word vectorizations.

Design

Numerous algorithms in Natural Language Processing (NLP) compute mathematical representations of words. These representations have shown to encode important semantic relationships and allow for more effective language modeling.

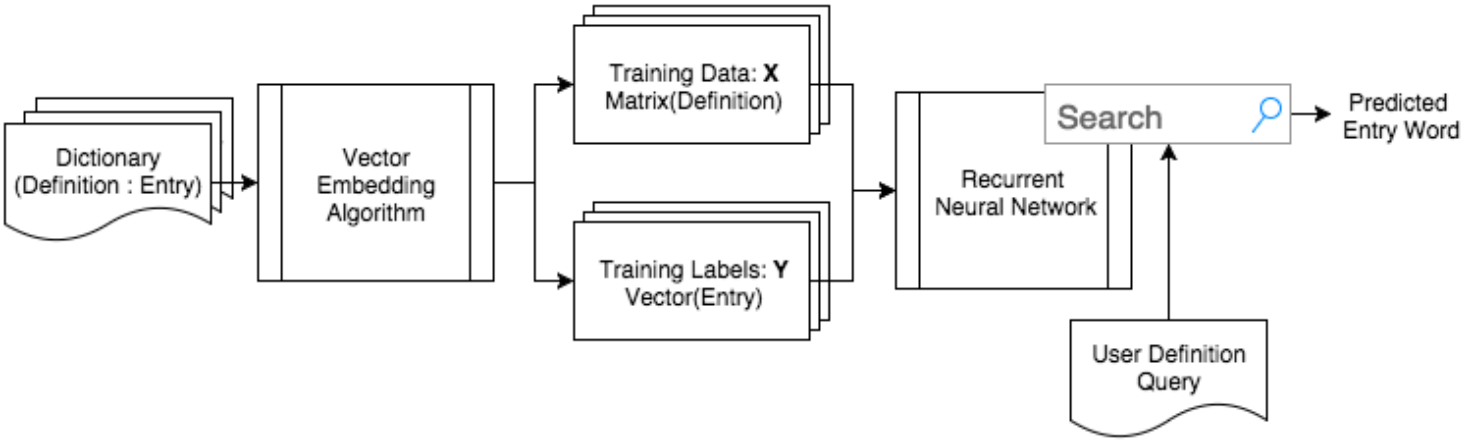
One methodology, *word2vec* [1], computes vector representations of words such that linear operations have semantic meanings. A common example is:

$$v(king) - v(man) + v(woman) \approx v(queen)$$

Where if you subtract the vector representing man from king and then add the vector woman, the result will be close to the representation of queen.

Examples like this have led us to question the extent of possible operations. Vector embeddings are commonly used in deep learning systems for a multitude of different tasks, for example sentiment analysis [2] and learning how to answer questions about written stories [3]. We will take these techniques and apply them to a new task: building a reverse dictionary.

The premise is to take the definitions of a dictionary and build a deep neural network that learns to predict the corresponding entry words. We expect the network will learn general enough definitions for words such that a user can query the dictionary with a definition and the reverse dictionary will be able to predict which word the user is thinking of. The overall design of the system is below:



We will take our training data (a set of definitions and their respective dictionary entries) and compute a vector representation of the words and defintions (a matrix of word vectors). We will then have numerical input for the network. The architecture of the network will be a recurrent neural network. We will train the network on the input data and attempt to minimize the euclidian distance between the predicted word vector and the true dictionary entry. Lastly, we will build a UI to allow users to query their own definitions. The system will predict possible dictionary entries and display them to the user.

Implementation

Training Data

We scraped multiple dictionaries to produce an an ecclectic dataset of dictionary entries. The hope was to provide more linguistic diversity for the neural network to train on. The dictionaries we used are the follow:

- Princeton University's WordNet [4]
- Oxford English Dictionary [5]
- Webster's Unabridged Dictionary [6]

For ease of training, we capped each of the definitions to 20 words. For defitions that were shorter than 20 words, we right padded them with a special padding token <PAD>.

Word Embeddings

Running these algorithms can be very time intensive, so numerous projects have surfaced to provide

precalculated embeddings. We utilized the embeddings from the *Polyglot* project [7].

The Polyglot project trained their models on the English Wikipedia corpus. They embedded a vocabulary set of 100,000 words into 64 dimensions.

We cross-referenced our set of dictionary entries with the Polyglot's vocabulary list because we only wanted to train on words that had an embedded vector representation. This resulted in a dataset of 338,637 dictionary entries for 62,663 unique words. We paritioned a third of that dataset and reserved it as a testing set.

Recurrent Neural Networks

One of the biggest advantages of Recurrent Neural Networks (RNN) is the ability to compute mathematical operations across sequences of vectors. This fact makes them perfect for computing natural language tasks, because understanding language is sequential in nature. For humans to understand that in the sentence "In France, we spoke their native language" the word *language* refers to *French* we need to at least remember reading the word *France*.

RNN have been able to perform even better through the development of Long Short-term Memory (LSTM) networks. On a higher level, these LSTM networks work by storing an internal cell state that 'remembers' previous information and uses that in future computation of the sequence (the mathematical theory behind LSTM networks is beyond the scope of this write-up [8]).

We used the Keras Python module to implement the reverse dictionary's RNN. Keras is a modular, quick and easy way for implementing neural networks. It It runs on top of the Theano deep learning package and provides multiple outlets for customizing the network. Our final implementation of the network is as follows:

- Input Layer (Input: 64 nodes)
- LSTM Layer (Output: 512 nodes)
- LSTM Layer (Output: 512 nodes)
- Fully Connected Layer with Tanh Activation (Output: 512 nodes)
- Fully Connected Layer with Linear Activation (Output: 64 nodes)

This model allowed for the network to discover any nonlinear interactions from the output of the LSTMs. Because of the limitations of the available Theano package (the HPC cluster only has Theano 0.7.0), we were not able to use Rectified Linear Units. The values of Polyglot's embeddings loosely ranged from -5 to 5. The tanh activation is limited to [-1,1] and sigmoid to [0,1], therefore we used a top layer with linear activations for the network to be able to compute the full range of embedding values.

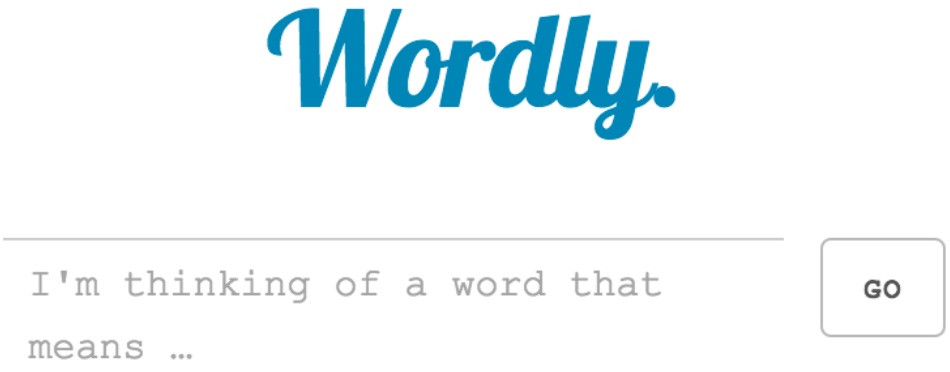
We evaluated the network with a Euclidian Distance objective function:

$$J(\theta) := \sqrt{(y_{pred} - y_{true})^2}$$

Where y_{pred} is the predicted dictionary entry vector and y_{true} is the vector embedding of the correct dictionary entry. Keras (on top of Theano) symbolically computes the gradients of the loss function with respect to the network parameters θ.

User Interface

Wordly's user interface is an attractive web-based platform designed for simplicity, elegance, and performance. It primarily consists of a simple textbox [for the user's *description*] and a "go" button.



This interface is designed for carefree ease-of-use, making its proudest features unobtrusive: the text input feild scales to fit the input, the GUI scales responsively, and the response/explanation feilds hide when unused.

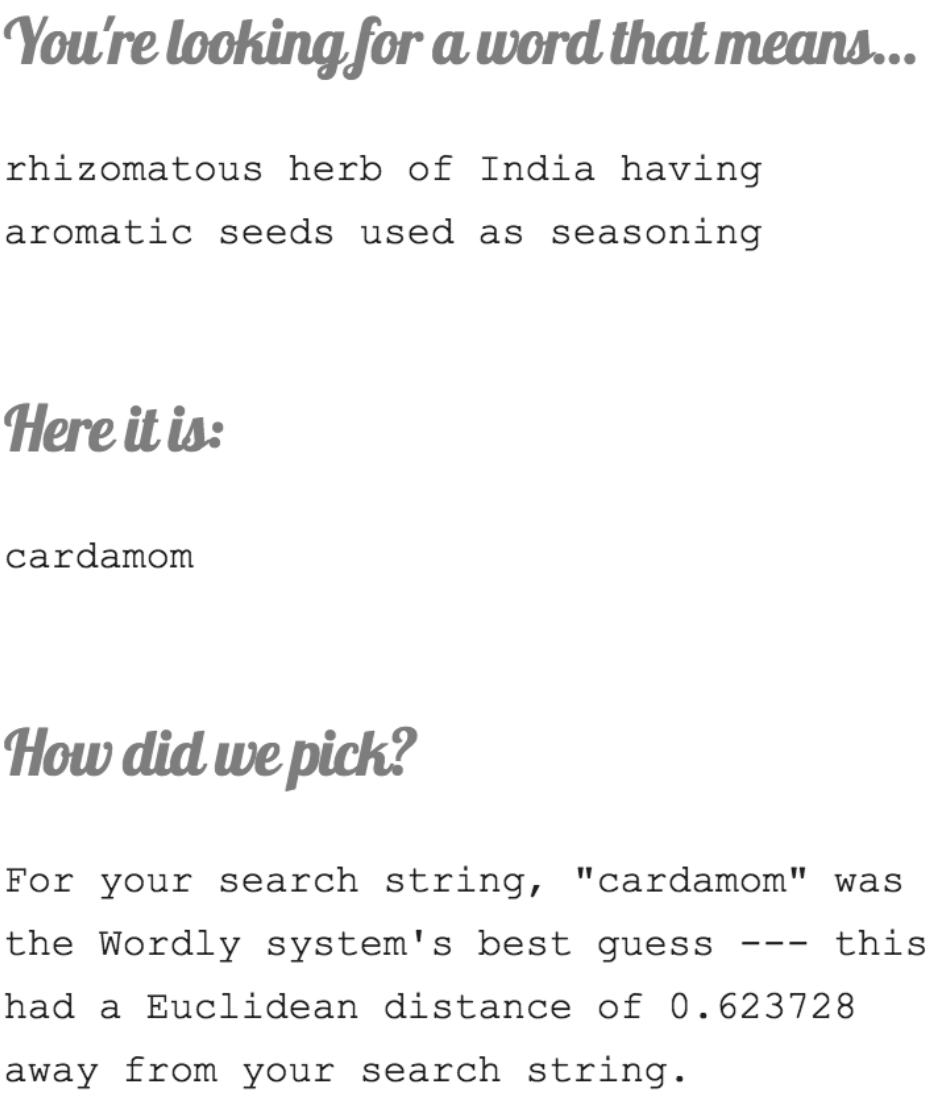
Wordly can be accessed here: http://54.152.167.250/Wordly

Despite the minimal design and simplicity of user interaction, we built the GUI on top of a full-fetured *express.js*[9] app that is deployed on *Amazon AWS*[10] cloud management. In addition to making the service publicly available, this decision enables us to run the RNN generated back-end python model live on the server.

The core of the Wordly system is constructed as a python object that is instantiated when the server starts. Following the Singleton programming pattern, this large, cumbersome model simply persists after it is created. Subsequent queries can be efficiently constructed thereafter.

When a user makes a query, the response is displayed below the search box in three sections: a cope of the

user's query, then the best choice word, then an explanation. The Explain function, another core feature of this project, is a statistical summary of the results from the RNN model's output.



UI Credits

We used a number of pre-existing libraries and technologies to make this project as full-featured, robust, and modular as possible. The packages and projects that we relied the most upon are:

- node.js [11]
- npm [12]
- nunjucks [13]

- CrowdSound [14]
- Skeleton.js [15]
- Autosize [16]
- Browserify [17]

Results

We ran a couple of experiments, trying a couple of different neural network architectures. The three models and the results are summarized in the table below:

Model 1	**Model 2**	**Model 3**
LSTM (512 Nodes)	LSTM (512 nodes)	LSTM (512 nodes)
LSTM (512 Nodes)	LSTM (512 Nodes)	LSTM (512 Nodes)
Linear Fully Connected (64 Nodes)	Tanh Fully Connceted (512 Nodes)	LSTM (512 Nodes)
	Linear Fully Connceted (64 Nodes)	
Avg. Euclidian Dist:	**Avg. Euclidian Dist:**	**Avg. Euclidian Dist:**
2.06332545841	2.12454035878	2.23333931759

Because of our limitations on computing power, we were unable to determine if any of these models converged. Therefore, to continue we choose to implement Model 1.

Unfortunately, an average euclidian distance of ~2.06 is not fairly impressive. Most word embeddings have a nearest neighbor in the range of ~1.3 units. Thus, if we are predicting with an error radius of 2.06 it will be difficult to have the correct word be the first choice. For this reason, we implement Wordly to return the 5 most likely dictionary entries.

We wanted to explore which words Wordly learned to predict well and which it could not understand. The following is the list of the 10 best predictions with their definitions:

Word	Definition	Distance
oxytocin	hormone secreted by the posterior pituitary gland (trade name Pitocin)	0.574378
cardamom	rhizomatous herb of India having aromatic seeds used as seasoning	0.623728
heparin	a polysaccharide produced in basophils (especially in the lung and liver) and that inhibits the activity of thrombin in coagulation of the blood	0.627694
Paleozoic	Of an era of geological time marked by the appearance of plants and animals, esp. Invertebrates.	0.630854
strychnine	formerly used as a stimulant	0.634796
Birmingham	A city in central England	0.636163
glassy	(of ceramics) having the surface made shiny and nonporous by fusing a vitreous solution to it	0.646284
innocently	in a naively innocent manner	0.647628
mistletoe	partially parasitic on beeches, chestnuts and oaks	0.651191
Daoism	philosophical system developed by Lao-tzu and Chuang-tzu advocating a simple honest life and noninterference with the course of natural events	0.655668

This is a list of words that are very specific in their defintions. When we run the definitions through Wordly we get the correct result.

However, many of these words have multiple definitions. For example cardamom has two definitions in our test set: "rhizomatous herb of India having aromatic seeds used as seasoning" and "aromatic seeds used as seasoning like cinnamon and cloves especially in pickles and barbecue sauces." Wordly is able to understand the first definition and predicts cardamom as the first predicted word. But Wordly's confidence dwindles for the second definition and chili replaces cardamon as the closest word.

Vector embeddings of two words are close if they are semantically similar. We hypothesize that the fact that Wordly learns very specific words is that their these word embeddings are distributed farther apart from more commonly used words. Therefore, it is easier for the RNN to learn weights that predict vectors substantially different from majority of the other words in the training set. If this hypothesis is true, we would expect that the model performs the worst on the most common words. Below is a sampling of the three words that Wordly performed the worst on:

Word	Definition	Distance
an	a unit of surface area equal to 100 square meters	7.569044
talk	idle gossip or rumor	7.138134
The	A word placed before nouns to limit or individualize their meaning.	6.267479

It logical that Wordly would have trouble learning the definitions for words humans eve have difficulty defining.

Currently, Wordly's performance is not optimal, yet it is incredibly promising. We hope to continuing working on the project in the future. There are multiple improvements we can make on the decision system to hopefully increase its accuracy.

Future Steps & Limitations

Vector Embeddings: These results show that Wordly was not able to generalize well for the multiple definitions of words. We suspect this is a problem with the shallwoness of the word embeddings. The Polyglot embeddings are only 64 units long. Other public vector embeddings, like Google's word2vec, at leas have a length of 300. The advantage of having a larger embedding space is that the representations of words can be more distributed across the space. If non-similar words have a larger distance, than Wordly can have a larger error radius for predictions.

Vocabulary: Looking at Wordly's results, we noticed that a large subset of the vocabulary was proper nouns. This is slightly counter-intuitive to the goal of the system. We wanted to create a reverse dictionary that represented the complexity and intricacies of human language, not a trivia-answering machine. The vocabulary list we used was due to the restraints of the Polyglot dataset. To improve the system, we would change the vocabulary to a more comprehensive list of words, with many of the proper nouns and entities removed.

Computational Power & Time: Running these RNN is a computationally intensive task. We were limited to using Yale University's High Performaning Clusters. The cluster requires running jobs in a queue system and so for the sake of fairness, jobs are not allowed to run ad infinitum. We were unable to determine whether any of the models converged and thus forced to choose the best model with the lowest validation error. Next steps would be to further refine the model and run them for enough time to obtain conclusive results.

Installation Guide

Theano

Theano has an easy installation for a vanilla version. You can use PyPI:

```
sudo pip install Theano
```

We used version `0.7.0`. Theano depends on the follow packages:

- numpy, scimpy
- gcc+

It is more difficult to unlock the GPU features of Theano and not neceesary to run our code. For reference, the GPU configuration and installation guide is here.

Keras

More thorough installation guide: http://keras.io/#installation. Keras depends on the following Python packages:

- numpy, scimpy
- pyyaml
- h5py

(each can be installed using conventional package managers, e.g. `pip`).

The easiest way to install Keras is through the PyPI:

```
sudo pip install keras
```

We used Keras version `0.3.0`

Wordly

We have submitted with this write up all of our scripts that have been used to scrape, filter, and process the data. Additionally, we have included the code to train a RNN. However, we spent days training our different models and therefore we do not expect you to do the same.

Our results have been packaged into a final module called `wordly.py`. Wordly.py contains a `Wordly` class that can start predicting dicitonary entries in two steps:

First, we instantiate and initialize the model:

```
from wordly import Wordly
model = Wordly()
model.initialize_model(weights = 'weights.hdf5', # Local Weights File
                       embeddings = 'embedding.pkl') # Local Embedding File
```

This requires two local files: `weights.hdf5` and `embedding.pkl`. The weights file must be custom to the model to be loaded. The embeddings file needs to have two unpackable values: `words` and `embeddings`. Words is the vocabulary list and embeddings is the corresponding vector embeddings.

Now, we can query the model

```
results = model.look_up('a city in central England', k = 5)
```

Results will be a list of length K containing the tuples `(PREDICTION, DISTANCE)` for the K closest predictions.

Wordly depends on the following packages:

- Keras
- H5PY
- Pickle
- Theano
- Numpy

Additionally, it depends on our script `process_data.py` for preprocessing sentences. Ensure to append the local directory of `process_data.py` to your `$PATH$` variable.

Appendix

Attached with this writeup are the following files:

- Dictionary Scraping Scripts:
  - CleanOPTED.py
  - CleanOxford.py
  - CleanWebsters.py
  - SearchWN.py
- Dictionary Processing Scripts:
  - process_data.py

        - Convert a given dictionary to their vector embeddings
    - filter_dicts.py
        - Given a vocabulary list, extract all of the entries of the dictionary and put it in one dictionary
    - extract_embeddings.py
        - (Test file) Extract embeddings can be used to get the embeddings from a large embedding binary file. This could be used for the Google word2vec embeddings.
- Training Recurrent Neural Network
    - wordly_rnn.py
- Run and Implement Wordly:
    - Wordly.py
    - weights.hdf5
    - embeddings.hdf5
- Corpus Scraping (Test)
    - StripHeaders
    - stripStopwords.pl
    - Combine
    - StripCode
    - ParseEmbeddingCorpus.py

---

1. Word2vec is not an actual algorithm but a framework for implementing the vectorization of words. Word2vec models implement either the skip-gram algorithm or Continuous Bag of Words (CBOW). Continuous Bag of Words attempts to predict a word from its context. Meanwhile, the skip-gram algorithm is the reverse; it attempts to predict the context from a given word. Increasing the probability of a word given another word is eqauted with decreasing the distance between these two words in vector space. Training these algorithms over large natural language corpouses can result in vectors that are able to perform semanticaly meaningfuly computations. See Google's Word2Vec for more information. ↩
2. LSTM Networks for Sentiment Analysis ↩
3. Facebook bAbI project ↩
4. Princeton University "About WordNet." WordNet. Princeton University. 2010. http://wordnet.princeton.edu ↩
5. Oxford Dictionaries. Oxford University Press, n.d. Web. 20 December 2015.

https://github.com/sujithps/Dictionary/blob/master/Oxford%20English%20Dictionary.txt ↩

6. Lawrence, Graham, comp. Webster's Unabridged Dictionary. N.p.: Project Gutenberg, 2009. Text file. https://ia600500.us.archive.org/zip_dir.php?path=/15/items/webstersunabridg29765gut.zip&formats=TEXT ↩

7. Polyglot: Distributed Word Representations for Multilingual NLP, Rami Al-Rfou, Bryan Perozzi, and Steven Skiena. In Proceedings Seventeenth Conference on Computational Natural Language Learning (CoNLL 2013). ↩

8. For further reading on how LSTMs work see Christopher Olah's blog on Understanding LSTM Networks and Andrej Kaparthy's The Unreasonable Effectiveness of Recurrent Neural Networks ↩

9. express.js is a framework that simplifies the creation of web-apps built in node.js ↩

10. Amazon AWS ↩

11. node.js is a full-featured system for writing server-side javascript code, and is the underlying language of our system. ↩

12. npm is the node.js package manager. ↩

13. nunjucks is a templating engine for node.js ↩

14. Base code, including all install files and basic webserver/app taken from code written for CrowdSound, a project for Yale's CPSC439 in 2015. It was created by Eli Block, Jack Feeny, Tim Follo, Jared Katzman, and Terin Patel-Williams. ↩

15. Skeleton is a piece of CSS boilerplate that helps to style our site. ↩

16. Autosize is a short script that makes the text box scale to input size. ↩

17. Browserify is a package that allows us to serve browser-side javascript (for autosize and other DOM manipulations) through express without messying our html templates. ↩