Multimodal Gaussian Word Embeddings for the purpose of Acronym Disambiguation

Cameron English

Abstract

Acronym disambiguation is a notoriously difficult problem in word-sense disambiguation due to the oft-polysemous nature of acronyms. Multimodal probabilistic word embeddings have shown great promise in natural language processing tasks concerned with words with multiple meanings. This paper will explore the usage of Gaussian mixture model-based word embeddings for the purpose of acronym disambiguation as opposed to typical unimodal probabilistic approaches or deterministic point vector approaches.

Computer Science
Under the supervision of David Read
Skidmore College
April 2018

Contents

Acknowledgments

I would like to give special thanks to Lisa Schermerhorn, Ben Turtel, Ben Mcmorran, Seung-won Hwang, Lukas Prediger, Roman Prokofyev, David Read, and Varun Kumar, without whom this work would never have seen the light of day.

1 Introduction

Acronym disambiguation is a subproblem in the greater subject area of word-sense disambiguation, a field that seeks to understand which of a given word's meanings is the one being chiefly called upon in context. For example, consider the word *mine*. *Mine* can function in multiple ways: as a possessive adjective, as a noun indicating the location of mineral extraction, as a noun indicating an explosive device placed underground, and as a verb related to these latter two uses, all with no change to the root word *mine* whatsoever. The average word is estimated to contain approximately 2.8 to meanings (Koichiro and Jun, 1989); acronyms exacerbate this problem wildly. The average acronym will contain around 12.5 distinct meanings, a problem compounded by the significantly higher introduction rate of new acronyms as compared to new words (*AcronymFinder*, 2018). The 2015 AcronymFinder database stored 4,779,434 definitions with an average 37 new entries added each day; while the statistics for the website stopped being updated at some juncture, this should nonetheless give an indicator of the rate of growth and relevancy of the problem.

1.1 Foundational Work

A great deal of effort has been poured into this problem, due largely to the prevalence of acronyms in scientific and medical fields. A variety of approaches have been employed: efforts revolving around the semantic and ontological data contained in the Unified Medical Language System knowledgebase provided powerful results but cannot be applied to the general domain for lack of a knowledgebase (Bodenreider, 2004). Subsequent attempts to tackle this problem center on natural language processing techniques such as syntactic feature examination, a bag-of-words approach, or topic modeling (HaCohen-Kerner et al., 2008), (Zhang et al., 2011). More recently, successful approaches have utilized the Word2Vec embedding schema to great effect (Li et al., 2015). This paper will examine the application of a variation of the skipgram model used in Word2Vec that employs a Gaussian mixture model-based approach as opposed to the standard deterministic point vectors. The usage of multimodal Gaussian embeddings has shown great promise with regard to problems of polysemy and entailment (Athiwaratkun and Gordon Wilson, 2017). This paper seeks to determine the efficacy of considering acronym disambiguation as specific case of a polysemy problem and to ascterain the feasibility of leveraging the benefits gained from the probabilistic approach.

1.2 Word2Vec

While the minutiae of the following topics lie outside the scope of this paper, a brief overview is in order to understand the subsequent model. Word2Vec operates by examining a large corpus of text and mapping words with similar meanings to nearby points in a vector space. These meanings are calculated based off of other words in a *word window* around the word being examined, typically chosen as a hyperparameter around 10. The advent of Word2Vec provided a huge boost in ability as a semantic-based approached compared to previous syntax-based approaches. The algorithm can use either a continuous bag-of-words approach or skip-gram based model; the latter is used for the model in this paper due to its superior results concerning words that appear infrequently. Word2Vec can employ several other key performance optimizations, most importantly negative sub-sampling. Negative sub-sampling devalues the most frequently appearing words in a corpus in a similar spirit as TF-IDF, as the semantic importance of words like *the* or *a* are typically of little interest.

1.3 Gaussian Distributions and Mixture Models

An alternative approach was proposed in 2014 by Vilnis and McCallum that chooses to represent words as Gaussian probability distributions instead of point vectors, learning the mean and covariance matrix from the data. An issue with this method is the unimodal approach employed; when faced with words (or acronyms) featuring a large amount of meanings, the single mode has difficulty giving a meaningful value as it will be "pulled" in separate directions by the different meanings of the word or be left centered around one meaning of the word to the detriment of the others. A Gaussian Mixture Model as proposed by Athiwaratkun and Wilson uses a multimodal approach to circumvent this issue, making it a logical choice to combat the highly polysemous nature of acronyms. The mathematical grounding for this is somewhat complex; those interested in seeing it should view the paper by Athiwaratkun and Wilson.

2 Methodology

This paper examines the application of the model of Athiwaratkun and Wilson to the problem of acronym disambiguation with an emphasis on higher K values for the multimodality of the model. The derivations for most of the equations can be found in that paper unless otherwise listed, as the *why* is outside the scope of this paper. The *how* is what we will be focusing upon, and these equations will thus be included as a means of formalization.

2.1 Mixture Model

$$f_w(\vec{x}) = \sum_{i=1}^{K} p_{w,i}\, \mathcal{N}\left[\vec{x}; \vec{\mu}_{w,i}, \Sigma_{w,i}\right]$$
$$= \sum_{i=1}^{K} \frac{p_{w,i}}{\sqrt{2\pi|\Sigma_{w,i}|}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu}_{w,i})^\top \Sigma_{w,i}^{-1}(\vec{x}-\vec{\mu}_{w,i})}$$

where $\sum_{i=1}^{K} p_{w,i} = 1$.

Figure 1: Eq 1 from AW. Describes the mixture model for word w.

Each acronym w is represented as a Gaussian mixture with K components. Each Gaussian component's mean vector of word w can represent one of the acronym's distinct meanings. For example, if w = 'MS' and $K = 2$, our results would likely create a mixture for Microsoft and one for multiple sclerosis. Increase K to 3 and we would perhaps see representation of a master of science degree. Mean vector $\overrightarrow{\mu}_{w,i}$ corresponds to the location of the i-th component of acronym w, similar to point embeddings in Word2Vec. $P_{w,i}$ shows the component probability (the mixture weight) while the sum $\Sigma_{w,i}$ corresponds to the covariance matrix that contains the uncertainty information.

2.2 Skipgram

$$P_n(w_i) = 1 - \sqrt{t/f(w_i)}$$
$$P_n(w_i) \propto U(w_i)^{3/4}$$

Figure 2: Top: Probability w_i will be discarded. Bottom: Unigram for negative context generation. (Mikolov et al. 2013)

Parameters are learned via continuous skip-gram approach, similar to that of Word2Vec, where word embeddings are trained to maximize the probability of observing a word (or in our case, acronym) given another nearby word. The subsampling and negative context words are used to remove frequent words in the same manner as Word2Vec, as less frequently observed are typically more semantically important. Even though this is being applied to a dataset that has already been cleaned, tokenized, and stripped of stopwords the benefits are still tangible. The mathematical reasoning for this lies largely outside the scope of the paper and can be readily viewed in Mikolov's publication.

2.3 Max-margin/Energy function/Similarity metric

The ranking objective used in this model can be seen in the following equation:

$$L_\Theta(w, c, c') = max(0, m - logE_\Theta(w, c) + logE_\Theta(w, c'))$$

Figure 3: Ranking objective Athiwaratkun and Gordon Wilson (2017))

The equation examines word pairs (*w*,*c*) and (*w*,*c'*). *w* is a word sampled from the corpus, *c* is one of the words contained in the word window *l*, and *c'* is some negative context word. For example, given the sentence "The soldier went AWOL after his last mission", could have *w* as "AWOL", the other other words in the sentence as context word values for *c* values, and *c'* will be a negative context word chosen at random like "hydrant". This procedure further follows the ideas put forth in (Mikolov et al., 2013). Simply put, it pushes the similarity of a word/acronym and its positive context higher than that of its negative context by a margin *m*. Energy-based just means a measure of similarity. It can be further improved upon by mini-batch stochastic gradient descent with respect to the parameters discussed earlier (mean vectors, mixture weight, and covariance matrix). The $logE_\Theta$ represents the energy, which will be covered shortly.

$$E(f, g) = \int f(x)g(x)dx = \langle f, g \rangle_{L2}$$

Figure 4: Expected Likelihood Kernel Athiwaratkun and Gordon Wilson (2017)

The AW paper makes use of a concept from (Jebara et al., 2004) known as an expected likelihood kernel. This kernel functions as a generalization of the inner product between distributions; recall that we need to not only convey the point similarity but also the *uncertainty*. Recall further that we are not simply interested in *E* but in *logE*, whose calculation is a little more involved:

For Gaussian mixtures f, g representing the words w_f, w_g, $f(x) = \sum_{i=1}^{K} p_i \mathcal{N}(x; \vec{\mu}_{f,i}, \Sigma_{f,i})$ and $g(x) = \sum_{i=1}^{K} q_i \mathcal{N}(x; \vec{\mu}_{g,i}, \Sigma_{g,i})$, $\sum_{i=1}^{K} p_i = 1$, and $\sum_{i=1}^{K} q_i = 1$, we find the log energy is

$$\log E_\theta(f, g) = \log \sum_{j=1}^{K} \sum_{i=1}^{K} p_i q_j e^{\xi_{i,j}}$$

where

$$\begin{aligned} \xi_{i,j} &\equiv \log \mathcal{N}(0; \vec{\mu}_{f,i} - \vec{\mu}_{g,j}, \Sigma_{f,i} + \Sigma_{g,j}) \\ &= -\frac{1}{2} \log \det(\Sigma_{f,i} + \Sigma_{g,j}) - \frac{D}{2} \log(2\pi) \\ &-\frac{1}{2}(\vec{\mu}_{f,i} - \vec{\mu}_{g,j})^\top (\Sigma_{f,i} + \Sigma_{g,j})^{-1} (\vec{\mu}_{f,i} - \vec{\mu}_{g,j}) \end{aligned}$$

Figure 5: Eq 2(top) and 3(bottom) (Athiwaratkun and Gordon Wilson, 2017)

In these equations, $\xi_{i,j}$ represents the partial log energy, capturing the *i*-th meaning of word w_f and the *j*-th meaning of word w_g. The first equation represents the sum of possible pairs of partial energies, weighted accordingly by the mixture probabilities p_i and g_j The term -1/2logdet($\Sigma_{f,i} + \Sigma_{q,j}$) acts as a regularizer keeping high covariances caused by large values for $\Sigma_{f,i}$ and $\Sigma_{g,j}$ in check. High energy values $E_\Theta(w, c)$ will occur when the component means of the representative vectors $\vec{\mu}_{f,i}$ $\vec{\mu}_{g,j}$ are similar. Intuitively, related concepts will lead to a high energy value. Over the course of training individual semantic representations will become more distinct, causing one of the terms of $\xi_{i,j}$ to become the choice for the most related semantic pair. The derivation of this equation lies outside the scope of this paper and can be found in the Appendix of the Athiwaratkun and Wilson paper; it will suffice for this paper to have a formal definition of the methodology.

2.4 Hyperparameters

The hyperparameters available for adjustment are number of mixtures K, word window size l, embedding size D, initial learning rate η, minimum number of occurrences to place a word in the vocabulary m, subsample threshold s, minibatch size b, and number of epochs to train.

One of the key areas this paper differs from the Athiwaratkun and Wilson (AW) paper is in its focus on higher values of K in the model (more modes). This is due to the very high average number of meanings for a given acronym when compared to a given word. The AW paper provides results primarily for K=2 and no results for K higher than 3 due to diminishing returns and increased computational complexity, a sensible conclusion given the average meanings per word being in the 2-3.5 range. This will be the most relevant hyperparameter moving forward.

2.5 Evalutation

The standard approach for evaluating word embeddings consists of using the cosine similarity between the two vectors. This metric holds true for the Gaussian embeddings used in this experiment. An additional metric being considered is the minimum Euclidean distance between vectors due to its involvement in the training metric.

3 Datasets and Testing

3.1 Training dataset

Two different datasets were used to train the models (along with various hyperparameter permutations and combinations applied to them). The first dataset is the concatenation of *UKWAC+Wackypedia* specified in the original Athiwaratkun paper. The second dataset used was a cleaned version of the English-language Wikipedia. The specific version used is the April 2nd, 2018 edition of enwiki-latest-pages-articles.xml.bz2 downloaded from the Wikimedia dumps. This was then parsed by WikiExtractor to remove Wikipedia formatting and reduce the texts to more "normal" articles. These articles were then further cleaned by a simple preprocessing script included in the Appendix that performs expected normalization, tokenization, lowercasing, punctuation removal, etc. Note that stemming and lemmatization are not applied due to their projected interference with the focus on acronym disambiguation. The reasoning for this was to test the ability of general scalability of the model as well as to examine if it had additional strengths regarding in-domain data (as is often the case with word embeddings).

3.2 Testing and Validation datasets

Due to the non-existence of large, labeled, acronym-dense datasets with a focus on polysemy, a custom testing dataset was engineered for this task, a sample of which is included in the Appendix. The other main testing datasets are portions of the MSH and ScienceWise testing sets mentioned in Profokyev and Li's papers. It is worth noting that despite the papers' claims to the contrary, these datasets are not currently publicly available and only a portion sufficient for testing but not training could be obtained. In order to ensure the model ran properly, further tests were made against the datasets included in the original Word2Gaussian paper. As the model will be extracting *all* words of sufficient frequency into the vocabulary, filtering of the acronyms is required. This is accomplished by cross-referencing a list of acronyms crawled from Wikipedia (around 800,000 in total, including duplicates).

3.3 Testing

Training was conducted on the *UKWAC + Wackypedia* and Wikipedia datasets with a variety of hyperparameters. NLP-standard word window size l=10, embedding size D=50, minibatch size b=128, sub-sample threshold s=0.0001, initial learning rate η=0.05 were found to be reasonably successful. Larger embedding size would have potential improvements to performance but was an impossibility due to memory limitations. This leaves the hyperparameter of interest, K. Testing was conducted for K=1,2,3,4,5. While higher values for K were initially desired, and indeed were at the heart of the experiment, testing showed this to be

perhaps less important than initially expected. Test results were compared to the off-the-shelf implementations of Word2vec (gensim) and GloVe (Stanford) fed the same training sets as the model for an unbiased comparison. Spearman correlation with regard to correct labeling is used as the examination standard.

4 Results

UKWAC+Wackypedia Model							
Test set	w2gm K=2	K=3	K=4	K=5	K=1	w2v	GloVe
ADT	61.1	65.3	60.0	**68.8**	54.3	67.7	63.3
MSH	**72.5**	70.1	68.4	69.8	66.2	60.6	64.7
ScienceWise	71.8	69.3	**74.4**	68.3	66.4	64.7	63.9

Wikipedia Model							
Test set	w2gm K=2	K=3	K=4	K=5	K=1	w2v	GloVe
ADT	76.9	80.3	**82.7**	81.4	69.1	76.8	69.4
MSH	**68.6**	64.6	63.5	64.9	60.9	58.3	62.9
Sciencewise	**73.8**	69.9	71.1	67.8	65.8	72.7	65.8

The value shown is that given by the expected likelihood kernel or the cosine similarity, whichever proved better (the cosine similarity in almost all cases but K=5 for both models, an interesting aside). The ADT test set refers to the "Acronym Disambiguation Testset" created for this experiment. The dataset was inspired by Wikipedia articles in an effort to showcase the power of in-domain embeddings while at the same time not simply re-using part of the training set. A qualitative view of the results in a 3D Tensorboard visualization provides expected results.

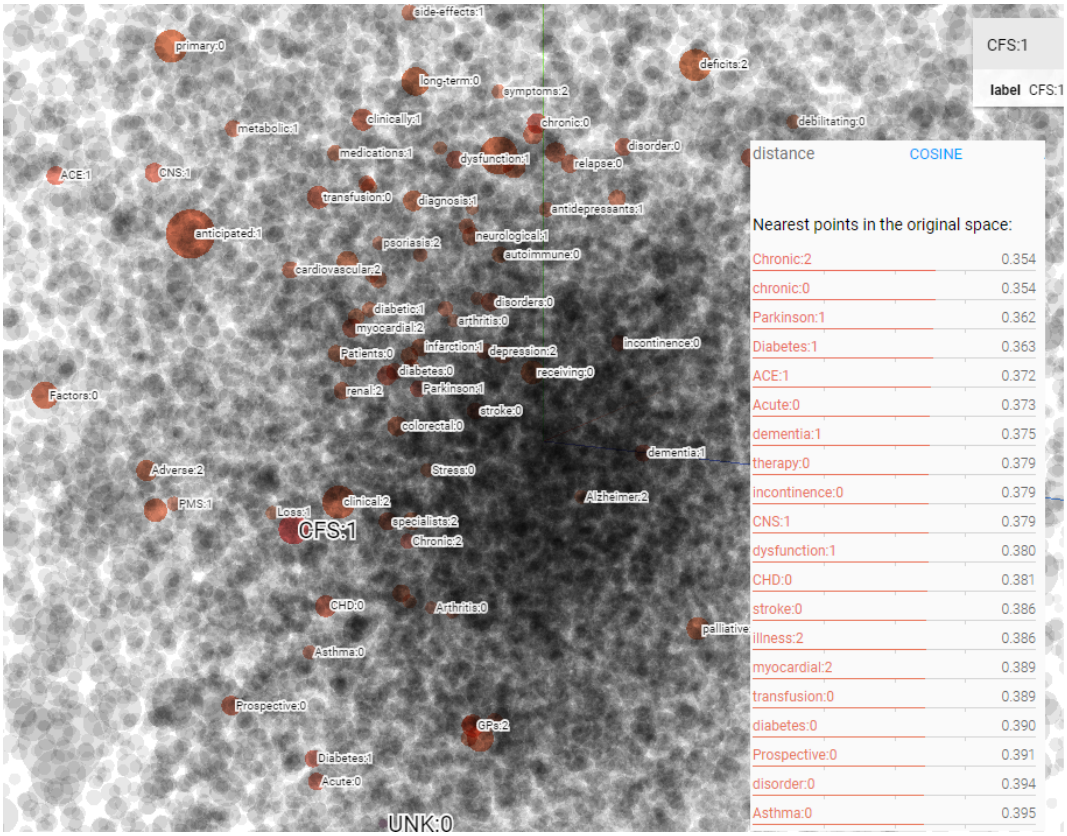

Figure 6: Snapshot of limited Tensorboard visualization for term a sense of "CFS" (Chronic Fatigue Syndrome)

5 Discussion

5.1 Discussion

The Gaussian mixture model performs well, standing toe-to-toe with its state-of-the-art peers. It generalizes effectively and benefits significantly from in-domain embeddings while not requiring a knowledgebase. It offers sizable improvements over the unimodal approach across the board. It does not, however, offer the desired massive improvements over a simple Word2Vec implementation. Further making the results difficult to interpret is the huge variation across K values; this could possibly be attributable to the ADT dataset being the only one offering an extreme focus on polysemy, but even that sees a drop between K=4 and K=5 in the Wikipedia model. This drop could, of course, be attributable to the probabilistic nature of the training, but it's difficult to pin down if that's the real reason. Other confusing results exist: why does word2vec perform so well on ScienceWise (outperforming all but one of the w2gm models)? Why is GloVe *so* consistent over the datasets? Why are the K=5 results so close when trained on $UKWAC+Wackypedia$ but so far apart on Wikipedia? At the very least, the multimodal model *does* outperform the unimodal model in almost every task. Note that the model in Li et al. was able to train on the entire MSH and ScienceWise datasets and use cross-validated in-domain embeddings to obtain such comparatively high accuracy.A possible explanation for the less-than-overwhelming results can be traced to the fundamental assumption of the Word2vec modeling schema: words are usually closely related to the words that immediately surround them. Perhaps with acronyms this assumption cannot be made with the same strength; anecdotal evidence from qualitative analysis of the two training datasets indicates that many times acronyms appear in sentences not very closely related to their meaning after they have been initially defined. This would point towards potential value in a weighting-based symptom that places more value on the defining sentence of the acronym, but this is purely speculative and a great deal of future experimentation would be required to support either portion of this conjecture.

5.2 Future Work

One of the first areas for future work is finding a way to allow for higher values of K to be input. Due to a dependency on protobuff, there is a hard limit of 2GB for serializing individual tensors because of the 32bit signed size. Practically speaking, this imposes a hard cap on the size of K allowed that would need a workaround. As testing showed, decent results were obtainable on polysemous acronyms even in the absence of double-digit K. It's difficult to tell if the higher value K would have any effect, but results from the ADT test set indicate at least the *possibility* of improvements in results by improving this area. Reducing the size of the tensor to slice by reducing hyperparameters such as the embedding size or word window have relatively little size reduction, and any benefit gained is nowhere near equal the loss of strength in the model from the reduction in these hyperparameters. Additional computational optimization could likely be obtained through sparse-vector optimization techniques, as not all acronyms will likely form higher-order modes, or form modes with such low values they could be rounded down without loss of modeling power in exchange for a great deal of performance improvement. Other future work could consist of the previously-mentioned examination of the fundamental Word2vec assumption (can acronyms truly be viewed simply as polysemous words) and potential workarounds. A final note is that longer training time (more epochs) might actually prove beneficial for the higher K models, as they were not entirely converged when training was stopped.

5.3 Conclusion

The efficacy of multimodal Gaussian-distribution-based embeddings for the purpose of acronym disambiguation was examined. The model showed itself to be comparable to other state-of-the-art approaches such as Word2vec and GloVe, but did not offer a massive improvement over them. The results contain a number of difficult to interpret pieces of information such as the variation over K values; while a neat improvement in performance in conjunction with an increase in K values, this was not the case. One aspect of the hypothesis that was confirmed was the improvement over unimodal Gaussian distributions across the board. Even these, however, were not always very large, or at least not large enough to make the enormously increased computational requirements easily justifiable. The murky optimism of the results, the ability to place as

well as the other state-of-the-art models, and the relative newness of probabilistic word embeddings lead to the conclusion that this is a topic worthy of further examination.

References

Abbreviations and acronyms dictionary, acronymfinder.com.

Ben Athiwaratkun and Andrew Gordon Wilson. Multimodal word distributions. In *Conference of the Association for Computational Linguistics (ACL)*, 2017.

Olivier Bodenreider. The unified medical language system (umls): integrating biomedical terminology. *Nucleic acids research*, 32(suppl1):267–270, 2004.

Y. HaCohen-Kerner, A. Kass, and A. Peretz. Combined one sense disambiguation of abbreviations. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers. Association for Computational Linguistics*, pages 61–64, 2008.

Tony Jebara, Risi Kondor, and Andrew Howard. Probability product kernels. *Journal of Machine Learning Research*, 5:819–844, 2004.

Matsuno Koichiro and Lu Jun. A word of a natural language has 2.718 different meanings or slightly more. *Bio Systems*, 22:301–4, 02 1989.

Chao Li, Lei Ji, and Jun Yan. Acronym disambiguation using word embedding. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4178–4179, 2015.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Neural Information Processing Systems*, 2013.

R Prokofyev, G Demartini, and A et al. Boyarsky. Ontology-based word sense disambiguation for scientific literature. *Advances in Information Retrieval*, pages 594–605, 2013.

Luke Vilnis and Andrew McCallum. Word representations via gaussian embedding. *CoRR abs/1412.6623*, 2014.

W. Zhang, Y. C. Sim, and J. et al. Su. Entity linking with effective acronym expansion, instance selection, and topic modeling. *IJCAI*, pages 1909–1914, 2011.

6 Appendix

6.1 Materials

- Python 2.7 (version important)
- TensorFlow 0.11.0rc1 with a modified Skipgram method to accommodate large datasets (version important) and its dependencies
- Additional Python packages: ggplot, glob, matplotlib, nltk, numpy, pandas
- Linux or Macintosh operating system for TensorFlow compatibility (0.11 does not run on Windows)
- 32 GB RAM; precise amount of requisite RAM was not determined, but 16 GB is insufficient to train on Wikipedia as-is
- WikiExtractor; https://github.com/attardi/wikiextractor . Freeware that parses a Wikipedia dump into more usable documents.
- While this experiment saw CPU-only training due to system constraints, GPU training utilizing the CUDA toolkit should speed things up immensely. It was not, however, tested here.
- Training took 1-2 weeks depending on the K value using the CPU-only method

6.2 Wikipedia training data sample

A not-so-random paragraph sampled from the cleaned Wikipedia corpus:
p versus np problem the p versus np problem is a major unsolved problem in computer science it asks whether every problem whose solution can be quickly verified technically verified in polynomial time can also be solved quickly again in polynomial time the underlying issues were first discussed in the 1950s in letters from john forbes nash jr to the national security agency and john von neumann the precise statement of the p versus np problem was introduced in 1971 by stephen cook in his seminal paper the complexity of theorem proving procedures and is considered by many to be the most important open problem in the field it is one of the seven millennium prize problems selected by the clay mathematics institute to carry a us 1000000 prize for the first correct solution

6.3 ADT dataset sample

This dataset was constructed from the scraped Wikipedia acronyms and page titles mentioned in section section 3.2. The dataset is a CSV of format Acronym, Label, Context word

MNS, mirror neuron system , Autism
BSMTE, bachelor of science in marine transportation and engineering , Maritime Academy of Asia
IWRG, international wrestling revolution group , Cachorro Mendoza
BRDF, bidirectional reflectance distribution function , Albedo
CBC, complete blood count , hemoglobin lepore syndrome

6.4 Code

```
    # word2gm_trainer.py taken with minimal adaptation from github.com/benathi/word2gm,
    # itself an adaptation of tensorflow's word2vec.py
    # (https://github.com/tensorflow/models/blob/master/tutorials/embedding/word2vec.py)
    # and is included to help transition ideas from concept and math to code
    # the project in its entirety (sans datasets) can be found at
    # https://github.com/CameronEnglish/word2gm
    # the remaining code was omitted from this paper due to space constraints;
    # its inclusion would add ~30 pages

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import sys
import threading
import time
import math
# Retrict to CPU only
os.environ["CUDA_VISIBLE_DEVICES"]=""

from six.moves import xrange  # pylint: disable=redefined-builtin

import numpy as np
import tensorflow as tf

from tensorflow.models.embedding import gen_word2vec as word2vec
```

```
flags = tf.app.flags

flags.DEFINE_string("save_path", None, "Directory to write the model and "
                    "training summaries. (required)")
flags.DEFINE_string("train_data", None, "Training text file. (required)")
flags.DEFINE_integer("embedding_size", 50, "The embedding dimension size.")
flags.DEFINE_integer(
    "epochs_to_train", 5,
    "Number of epochs to train. Each epoch processes the training data once "
    "completely.")
flags.DEFINE_float("learning_rate", 0.2, "Initial learning rate.")

flags.DEFINE_integer("batch_size", 256,
                     "Number of training examples processed per step "
                     "(size of a minibatch).")
flags.DEFINE_integer("concurrent_steps", 12,
                     "The number of concurrent training steps.")
flags.DEFINE_integer("window_size", 5,
                     "The number of words to predict to the left and right "
                     "of the target word.")
flags.DEFINE_integer("min_count", 5,
                     "The minimum number of word occurrences for it to be "
                     "included in the vocabulary.")
flags.DEFINE_float("subsample", 1e-3,
                   "Subsample threshold for word occurrence. Words that appear "
                   "with higher frequency will be randomly down-sampled. Set "
                   "to 0 to disable.")
flags.DEFINE_integer("statistics_interval", 5,
                     "Print statistics every n seconds.")
flags.DEFINE_integer("summary_interval", 5,
                     "Save training summary to file every n seconds (rounded "
                     "up to statistics interval).")
flags.DEFINE_integer("checkpoint_interval", 600,
                     "Checkpoint the model (i.e. save the parameters) every n "
                     "seconds (rounded up to statistics interval).")
flags.DEFINE_integer("num_mixtures", 2,
                     "Number of mixture component for Mixture of Gaussians")
flags.DEFINE_boolean("spherical", False,
                     "Whether the model should be spherical of diagonal"
                     "The default is spherical")

flags.DEFINE_float("var_scale", 0.05, "Variance scale")

flags.DEFINE_boolean("ckpt_all", False, "Keep all checkpoints"
                     "(Warning: This requires a large amount of disk space).")

flags.DEFINE_float("norm_cap", 3.0,
                   "The upper bound of norm of mean vector")
flags.DEFINE_float("lower_sig", 0.02,
                   "The lower bound for sigma element-wise")
flags.DEFINE_float("upper_sig", 5.0,
                   "The upper bound for sigma element-wise")
flags.DEFINE_float("mu_scale", 1.0,
                  "The average norm will be around mu_scale")
```

```
flags.DEFINE_float("objective_threshold", 1.0,
                   "The threshold for the objective")

flags.DEFINE_boolean("adagrad", False,
                   "Use Adagrad optimizer instead")

flags.DEFINE_float("loss_epsilon", 1e-4,
                   "epsilon parameter for loss function")

flags.DEFINE_boolean("constant_lr", False,
                   "Use constant learning rate")

flags.DEFINE_boolean("wout", False,
                   "Whether we would use a separate wout")

flags.DEFINE_boolean("max_pe", False,
                   "Using maximum of partial energy instead of the sum")

flags.DEFINE_integer("max_to_keep", 5,
                     "The maximum number of checkpoint files to keep")

flags.DEFINE_boolean("normclip", False,
                     "Whether to perform norm clipping (very slow)")

FLAGS = flags.FLAGS


class Options(object):
  """Options used by our Word2MultiGauss model."""

  def __init__(self):
    # Model options.

    # Embedding dimension.
    self.emb_dim = FLAGS.embedding_size

    # Training options.
    # The training text file.
    self.train_data = FLAGS.train_data

    # The initial learning rate.
    self.learning_rate = FLAGS.learning_rate

    # Number of epochs to train. After these many epochs, the learning
    # rate decays linearly to zero and the training stops.
    self.epochs_to_train = FLAGS.epochs_to_train

    # Concurgnt training steps.
    self.concurrent_steps = FLAGS.concurrent_steps

    # Number of examples for one training step.
    self.batch_size = FLAGS.batch_size
```

```
# The number of words to predict to the left and right of the target word.
self.window_size = FLAGS.window_size

# The minimum number of word occurrences for it to be included in the
# vocabulary.
self.min_count = FLAGS.min_count

# Subsampling threshold for word occurrence.
self.subsample = FLAGS.subsample

# How often to print statistics.
self.statistics_interval = FLAGS.statistics_interval

# How often to write to the summary file (rounds up to the nearest
# statistics_interval).
self.summary_interval = FLAGS.summary_interval

# How often to write checkpoints (rounds up to the nearest statistics
# interval).
self.checkpoint_interval = FLAGS.checkpoint_interval

# Where to write out summaries.
self.save_path = FLAGS.save_path

#####################################
self.num_mixtures = FLAGS.num_mixtures # incorporated. needs testing

# upper bound of norm of mu
self.norm_cap = FLAGS.norm_cap

# element-wise lower bound for sigma
self.lower_sig = FLAGS.lower_sig

# element-wise upper bound for sigma
self.upper_sig = FLAGS.upper_sig

# whether to use spherical or diagonal covariance
self.spherical = FLAGS.spherical   ## default to False please

self.var_scale = FLAGS.var_scale

self.ckpt_all = FLAGS.ckpt_all

self.mu_scale = FLAGS.mu_scale

self.objective_threshold = FLAGS.objective_threshold

self.adagrad = FLAGS.adagrad

self.loss_epsilon = FLAGS.loss_epsilon

self.constant_lr = FLAGS.constant_lr

self.wout = FLAGS.wout
```

```
    self.max_pe = FLAGS.max_pe

    self.max_to_keep = FLAGS.max_to_keep

    self.normclip = FLAGS.normclip

## value clipping
    self.norm_cap = FLAGS.norm_cap
    self.upper_sig = FLAGS.upper_sig
    self.lower_sig = FLAGS.lower_sig

class Word2GMtrainer(object):

  def __init__(self, options, session):
    self._options = options
    # Ben A: print important opts
    opts = options
    print('------------------------------------------------------------')
    print('Train data {}'.format(opts.train_data))
    print('Norm cap {} lower sig {} upper sig {}'.format(opts.norm_cap,
      opts.lower_sig, opts.upper_sig))
    print('mu_scale {} var_scale {}'.format(opts.mu_scale, opts.var_scale))
    print('Num Mixtures {} Spherical Mode = {}'.format(opts.num_mixtures, opts.spherical))
    print('Emb dim {}'.format(opts.emb_dim))
    print('Epochs to train {}'.format(opts.epochs_to_train))
    print('Learning rate {} // constant {}'.format(opts.learning_rate, opts.constant_lr))
    print('Using a separate Wout = {}'.format(opts.wout))
    print('Subsampling rate = {}'.format(opts.subsample))
    print('Using Max Partial Energy Loss = {}'.format(opts.max_pe))
    print('Loss Epsilon = {}'.format(opts.loss_epsilon))
    print('Saving results to = {}'.format(options.save_path))
    print('------------------------------------------------------------')

    self._session = session
    self._word2id = {}
    self._id2word = []
    self.build_graph()
    self.save_vocab()

  def optimize(self, loss):
    """Build the graph to optimize the loss function."""

    # Optimizer nodes.
    # Linear learning rate decay.
    opts = self._options
    if opts.constant_lr:
      self._lr = tf.constant(opts.learning_rate)
    else:
      words_to_train = float(opts.words_per_epoch * opts.epochs_to_train)
      lr = opts.learning_rate * tf.maximum(
         0.0001, 1.0 - tf.cast(self._words, tf.float32) / words_to_train)
      self._lr = lr
    optimizer = tf.train.GradientDescentOptimizer(self._lr)
```

```
    train = optimizer.minimize(loss,
                               global_step=self.global_step,
                               gate_gradients=optimizer.GATE_NONE)
    self._train = train

  def optimize_adam(self, loss):
    # deprecated
    opts = self._options
    # use automatic decay of learning rate in Adam
    self._lr = tf.constant(opts.learning_rate)
    self.adam_epsilon = opts.adam_epsilon
    optimizer = tf.train.AdamOptimizer(self._lr, epsilon=self.adam_epsilon)
    train = optimizer.minimize(loss, global_step=self.global_step,
                                          gate_gradients=optimizer.GATE_NONE)
    self._train = train

  def optimize_adagrad(self, loss):
    print('Using Adagrad optimizer')
    opts = self._options
    if opts.constant_lr:
      self._lr = tf.constant(opts.learning_rate)
    else:
      words_to_train = float(opts.words_per_epoch * opts.epochs_to_train)
      lr = opts.learning_rate * tf.maximum(
         0.0001, 1.0 - tf.cast(self._words, tf.float32) / words_to_train)
      self._lr = lr
    optimizer = tf.train.AdagradOptimizer(self._lr)
    train = optimizer.minimize(loss,
                               global_step=self.global_step,
                               gate_gradients=optimizer.GATE_NONE)
    self._train = train


  def calculate_loss(self, word_idxs, pos_idxs):
    # This is two methods in one (forward and nce_loss)
    self.global_step = tf.Variable(0, name="global_step")
    opts = self._options
    ###########################################################
    # the model parameters
    vocabulary_size = opts.vocab_size
    embedding_size = opts.emb_dim
    batch_size = opts.batch_size

    norm_cap = opts.norm_cap
    lower_sig = opts.lower_sig
    upper_sig = opts.upper_sig

    self.norm_cap = norm_cap
    self.lower_logsig = math.log(lower_sig)
    self.upper_logsig = math.log(upper_sig)

    num_mixtures = opts.num_mixtures
    spherical = opts.spherical
    objective_threshold = opts.objective_threshold
```

```
# the model parameters
mu_scale = opts.mu_scale*math.sqrt(3.0/(1.0*embedding_size))
mus = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures, embedding_size], -
if opts.wout:
  mus_out = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures, embedding_siz
# This intialization makes the variance around 1
var_scale = opts.var_scale
logvar_scale = math.log(var_scale)
print('mu_scale = {} var_scale = {}'.format(mu_scale, var_scale))
if spherical:
  logsigs = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures,1],
                                          logvar_scale, logvar_scale), name='sigma')
  if opts.wout:
    logsigs_out = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures,1],
                                          logvar_scale, logvar_scale), name='sigma_out

else:
  logsigs = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures, embedding_siz
                                          logvar_scale, logvar_scale), name='sigma')
  if opts.wout:
    logsigs_out = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures, embedd
                                          logvar_scale, logvar_scale), name='sigma_out

mixture = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures], 0, 0), name='m
if opts.wout:
  mixture_out = tf.Variable(tf.random_uniform([vocabulary_size, num_mixtures], 0, 0), m

if not opts.wout:
  mus_out = mus
  logsigs_out = logsigs
  mixture_out = mixture

zeros_vec = tf.zeros([batch_size], name='zeros')
self._mus = mus
self._logsigs = logsigs

labels_matrix = tf.reshape(
    tf.cast(pos_idxs,
            dtype=tf.int64),
    [opts.batch_size, 1])
# Negative sampling.
neg_idxs, _, _ = (tf.nn.fixed_unigram_candidate_sampler(
    true_classes=labels_matrix,
    num_true=1,
    num_sampled=opts.batch_size, # Use 1 negative sample per positive sample
    unique=True,
    range_max=opts.vocab_size,
    distortion=0.75,
    unigrams=opts.vocab_counts.tolist(), name='neg_idxs'))
self._neg_idxs = neg_idxs

def log_energy(mu1, sig1, mix1, mu2, sig2, mix2):
  ### need to pass mix that's compatible!
```

```
  def partial_logenergy(cl1, cl2):
    m1 = mu1[:,cl1,:]
    m2 = mu2[:,cl2,:]
    s1 = sig1[:,cl1,:]
    s2 = sig2[:,cl2,:]
    with tf.name_scope('partial_logenergy') as scope:
      _a = tf.add(s1, s2) # should be do max add for stability?
      epsilon = opts.loss_epsilon

      if spherical:
        logdet = embedding_size*tf.log(epsilon + tf.squeeze(_a))
      else:
        logdet = tf.reduce_sum(tf.log(epsilon + _a), reduction_indices=1, name='logdet
      ss_inv = 1./(epsilon + _a)
      diff = tf.sub(m1, m2)
      exp_term = tf.reduce_sum(diff*ss_inv*diff, reduction_indices=1, name='expterm')
      pe = -0.5*logdet - 0.5*exp_term
      return pe

  with tf.name_scope('logenergy') as scope:
    log_e_list = []
    mix_list = []
    for cl1 in xrange(num_mixtures):
      for cl2 in xrange(num_mixtures):
        log_e_list.append(partial_logenergy(cl1, cl2))
        mix_list.append(mix1[:,cl1]*mix2[:,cl2])
    log_e_pack = tf.pack(log_e_list)
    log_e_max = tf.reduce_max(log_e_list, reduction_indices=0)

    if opts.max_pe:
      # Ben A: got this warning for max_pe
      # UserWarning:
      # Convering sparse IndexedSlices to a dense Tensor of unknown shape. This may cor
      log_e_argmax = tf.argmax(log_e_list, dimension=0)
      log_e = log_e_max*tf.gather(mix_list, log_e_argmax)
    else:
      mix_pack = tf.pack(mix_list)
      log_e = tf.log(tf.reduce_sum(mix_pack*tf.exp(log_e_pack-log_e_max), reduction_in
      log_e += log_e_max
    return log_e


def Lfunc(word_idxs, pos_idxs, neg_idxs):
  with tf.name_scope('LossCal') as scope:
    mu_embed = tf.nn.embedding_lookup(mus, word_idxs, name='MuWord')
    mu_embed_pos = tf.nn.embedding_lookup(mus_out, pos_idxs, name='MuPos')
    mu_embed_neg = tf.nn.embedding_lookup(mus_out, neg_idxs, name='MuNeg')
    sig_embed = tf.exp(tf.nn.embedding_lookup(logsigs, word_idxs), name='SigWord')
    sig_embed_pos = tf.exp(tf.nn.embedding_lookup(logsigs_out, pos_idxs), name='SigPos
    sig_embed_neg = tf.exp(tf.nn.embedding_lookup(logsigs_out, neg_idxs), name='SigNeg

    mix_word = tf.nn.softmax(tf.nn.embedding_lookup(mixture, word_idxs), name='MixWord
    mix_pos = tf.nn.softmax(tf.nn.embedding_lookup(mixture_out, pos_idxs), name='MixPos
```

```
        mix_neg = tf.nn.softmax(tf.nn.embedding_lookup(mixture_out, neg_idxs), name='MixNeg

        epos = log_energy(mu_embed, sig_embed, mix_word, mu_embed_pos, sig_embed_pos, mix_p
        eneg = log_energy(mu_embed, sig_embed, mix_word, mu_embed_neg, sig_embed_neg, mix_n
        loss_indiv = tf.maximum(zeros_vec, objective_threshold - epos + eneg, name='Calcula
        loss = tf.reduce_mean(loss_indiv, name='AveLoss')

        return loss

    loss = Lfunc(word_idxs, pos_idxs, neg_idxs)
    tf.scalar_summary('loss', loss)

    return loss

  def clip_ops_graph(self, word_idxs, pos_idxs, neg_idxs):
    def clip_val_ref(embedding, idxs):
      with tf.name_scope('clip_val'):
        to_update = tf.nn.embedding_lookup(embedding, idxs)
        to_update = tf.maximum(self.lower_logsig, tf.minimum(self.upper_logsig, to_update))
        return tf.scatter_update(embedding, idxs, to_update)

    def clip_norm_ref(embedding, idxs):
      with tf.name_scope('clip_norm_ref') as scope:
        to_update = tf.nn.embedding_lookup(embedding, idxs)
        to_update = tf.clip_by_norm(to_update, self.norm_cap, axes=2)
        return tf.scatter_update(embedding, idxs, to_update)

    clip1 = clip_norm_ref(self._mus, word_idxs)
    clip2 = clip_norm_ref(self._mus, pos_idxs)
    clip3 = clip_norm_ref(self._mus, neg_idxs)
    clip4 = clip_val_ref(self._logsigs, word_idxs)
    clip5 = clip_val_ref(self._logsigs, pos_idxs)
    clip6 = clip_val_ref(self._logsigs, neg_idxs)

    return [clip1, clip2, clip3, clip4, clip5, clip6]

  def build_graph(self):
    """Build the graph for the full model."""
    opts = self._options
    # The training data. A text file.
    (words, counts, words_per_epoch, self._epoch, self._words, examples,
     labels) = word2vec.skipgram(filename=opts.train_data,
                                 batch_size=opts.batch_size,
                                 window_size=opts.window_size,
                                 min_count=opts.min_count,
                                 subsample=opts.subsample)
    (opts.vocab_words, opts.vocab_counts,
     opts.words_per_epoch) = self._session.run([words, counts, words_per_epoch])
    opts.vocab_size = len(opts.vocab_words)
    print("Data file: ", opts.train_data)
    print("Vocab size: ", opts.vocab_size - 1, " + UNK")
    print("Words per epoch: ", opts.words_per_epoch)
    self._examples = examples
    self._labels = labels
```

```
    self._id2word = opts.vocab_words
    for i, w in enumerate(self._id2word):
      self._word2id[w] = i
    loss = self.calculate_loss(examples, labels)
    self._loss = loss

    if opts.normclip:
      self._clip_ops = self.clip_ops_graph(self._examples, self._labels, self._neg_idxs)

    if opts.adagrad:
      print("Using Adagrad as an optimizer!")
      self.optimize_adagrad(loss)
    else:
      # Using Standard SGD
      self.optimize(loss)

    tf.scalar_summary('learning rate', self._lr)

    # Properly initialize all variables.
    self.check_op = tf.add_check_numerics_ops()

    tf.initialize_all_variables().run()

    try:
      print('Try using saver version v2')
      self.saver = tf.train.Saver(write_version=tf.train.SaverDef.V2, max_to_keep = opts.m
    except:
      print('Default to saver version v1')
      self.saver = tf.train.Saver(max_to_keep=opts.max_to_keep)

  def save_vocab(self):
    """Save the vocabulary to a file so the model can be reloaded."""
    opts = self._options
    with open(os.path.join(opts.save_path, "vocab.txt"), "w") as f:
      for i in xrange(opts.vocab_size):
        vocab_word = tf.compat.as_text(opts.vocab_words[i]).encode("utf-8")
        f.write("%s %d\n" % (vocab_word,
                             opts.vocab_counts[i]))

  def _train_thread_body(self):
    initial_epoch, = self._session.run([self._epoch])
    while True:
      # This is where the optimizer that minimizes loss (self._train) is run
      if not self._options.normclip:
        _, epoch = self._session.run([self._train, self._epoch])
      else:
        _, epoch, _ = self._session.run([self._train, self._epoch, self._clip_ops])
      if epoch != initial_epoch:
        break

  def train(self):
    """Train the model."""
    opts = self._options
    initial_epoch, initial_words = self._session.run([self._epoch, self._words])
```

```
    summary_op = tf.merge_all_summaries()
    summary_writer = tf.train.SummaryWriter(opts.save_path + "/epoch_%04d" % initial_epoch
    workers = []
    for _ in xrange(opts.concurrent_steps):
      t = threading.Thread(target=self._train_thread_body)
      t.start()
      workers.append(t)
    last_words, last_time, last_summary_time = initial_words, time.time(), 0
    last_checkpoint_time = 0
    step_manual = 0

    while True:
      time.sleep(opts.statistics_interval)  # Reports our progress once a while.
      (epoch, step, loss, words, lr) = self._session.run(
          [self._epoch, self.global_step, self._loss, self._words, self._lr])
      now = time.time()
      last_words, last_time, rate = words, now, (words - last_words) / (
          now - last_time)
      print("Epoch %4d Step %8d: lr = %5.3f loss = %6.2f words/sec = %8.0f\r" %
            (epoch, step, lr, loss, rate), end="")
      sys.stdout.flush()
      if now - last_summary_time > opts.summary_interval:
        summary_str = self._session.run(summary_op)
        summary_writer.add_summary(summary_str, step)
        last_summary_time = now
      if now - last_checkpoint_time > opts.checkpoint_interval:
        self.saver.save(self._session,
                        os.path.join(opts.save_path, "model.ckpt"),
                        global_step=step.astype(int))
        last_checkpoint_time = now
      if epoch != initial_epoch:
        break
      step_manual += 1

    # save the model after each epoch
    self.saver.save(self._session,
        os.path.join(opts.save_path, "model_epoch_%04d.ckpt" % initial_epoch),
        global_step=step.astype(int))

    for t in workers:
      t.join()
    return epoch

def _start_shell(local_ns=None):
  # An interactive shell is useful for debugging/development.
  import IPython
  user_ns = {}
  if local_ns:
    user_ns.update(local_ns)
  user_ns.update(globals())
  IPython.start_ipython(argv=[], user_ns=user_ns)

def main(_):
  if not FLAGS.train_data or not FLAGS.save_path:
```

```
    print("--train_data and --save_path must be specified.")
    sys.exit(1)
  if not os.path.exists(FLAGS.save_path):
    print('Creating new directory', FLAGS.save_path)
    os.makedirs(FLAGS.save_path)
  else:
    print('The directory already exists', FLAGS.save_path)
  opts = Options()
  print('Saving results to {}'.format(opts.save_path))
  with tf.Graph().as_default(), tf.Session() as session:
    with tf.device("/cpu:0"):
      model = Word2GMtrainer(opts, session)
    for _ in xrange(opts.epochs_to_train):
      model.train()
    # Perform a final save.
    model.saver.save(session,
                     os.path.join(opts.save_path, "model.ckpt"),
                     global_step=model.global_step)

if __name__ == "__main__":
  tf.app.run()
```