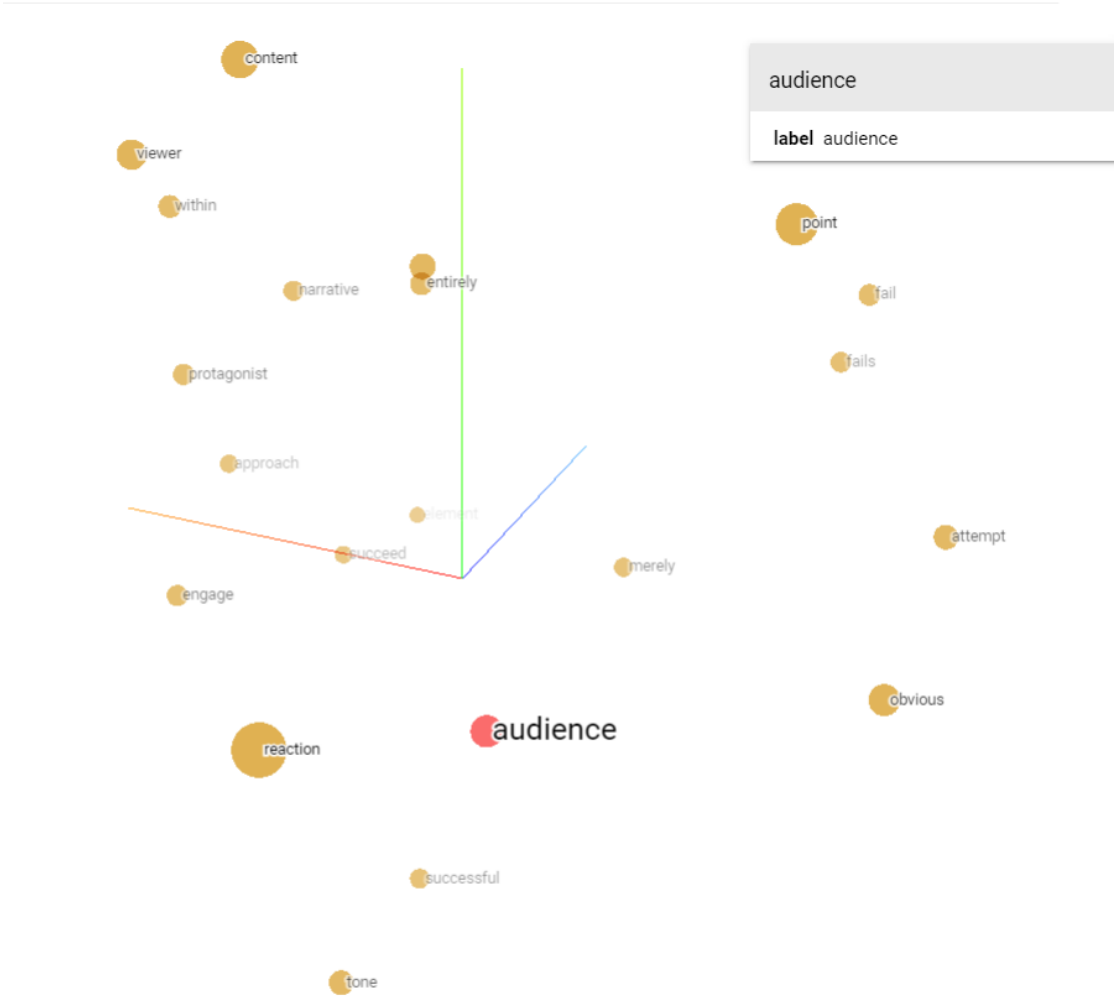
WORD EMBEDDINGS: A Comparison between Non-Neural Net and Neural Net Models



The word embedding for “audience” using TensorFlow’s Embeddings Projector (and a non-neural net approach) from a vocabulary of 20000 words with only 20 neighbors selected. The word vectors are 50 dimensional.

Mentor: Dr. Cheryl Fillekes

Intern: Christopher F. S. Maligec

Period: June – August, 2022

Contents

Abstract

Over the past decade deep learning neural net methods have come to dominate the NLP field and, in particular, the computation of word embeddings (real-valued vector representations of words) due to their greater speed over traditional methods such as LSA (Latent Semantic Analysis). However, these methods lack the transparency and explicability of the latter as it is not known how the neural net layers actually perform calculations. This paper compares the traditional non-neural net method using SVD against the neural net one of TensorFlow. In terms of efficiency, it has been found that the deep learning method of TensorFlow grows about 7.8 times faster than the Truncated SVD method used by scikit-learn. The type of SVD method is also relevant as TensorFlow's Thin SVD method is at least about $O(n^2)$ as opposed to scikit-learn's Truncated SVD, which is about $O(n)$. In terms of effectiveness, for a word vector dimensionality of 50, the two approaches are similar. Furthermore, two types of cosine similarity measurements for word vectors are compared: the standard one and one using a polynomial kernel. The two methods produced dissimilar results.

Keywords: word embeddings, SVD, deep learning, TensorFlow, scikit-learn, polynomial kernel

Introduction: Word Embeddings

Word embeddings are words that are mapped to real-valued vectors, so that these vectors encode the "meaning" of the words. As a result, words with similar "meanings" have vectors that are closer together in the vector space. Embeddings are of great value as they are essential to Natural Language Processing (NLP) tasks such as syntactic parsing and sentiment analysis. Methods to generate the mappings include dimensionality reduction, neural networks, and probabilistic models, among others (Wikipedia, Word embedding).

The vector spaces produced are high dimensional, very sparse, and difficult to process. There are methods to reduce the dimensionality from linear algebra such as Singular Value Decomposition (SVD - see further below), which led to Latent Semantic Analysis in the late 80s (Wikipedia, Latent semantic analysis).

Since 2005, neural net methods have come to dominate instead of the linear algebraic and probabilistic approaches. Around 2010, advances in the quality of vectors and the training speed of the models led to the adoption of this approach by researchers, and in 2013, word2vec could train vector space models faster than before (Wikipedia, Word embedding). However, the main problem with neural nets is that they are essentially black boxes in terms of how they work, so that a more traditional approach might be favored as it would be more explicable and transparent (Fillekes, personal communication).

This report focuses on a comparison of the non-neural net vs. neural net approaches to embeddings. Part I presents the algorithms to be compared. Parts II and III deal with efficiency, both in theory and in terms of running time results. Parts IV and V deal with effectiveness, both in theory and in terms of quality, i.e., word vectors with similar meanings should be closer together.

PART I: Algorithms

1. Algorithms

There are three word-embedding algorithms being compared in this report:

- sk: uses scikit-learn
- st: uses a TensorFlow non-deep learning approach
- dp: uses TensorFlow deep learning (neural net)

Link to code:

https://github.com/cfilleke/IBM-Intern-Projects/tree/main/TensorFlow/Embeddings

Each algorithm starts out with a vectorizer that takes in textual data and outputs a numerical encoding.

- For sk and st the vectorizer outputs word embeddings in terms of tf-idf values (float)
- For dp the vectorizer outputs integer encodings which will be used in an embeddings layer to produce the word embeddings.

As all three make use of a vectorizer, the part where they differ is

- For sk and st SVD (Singular Value Decomposition), which is used to reduce the dimensionality of the very sparse tf-idf matrices produced by their vectorizers. These reduced matrices will be the final versions of the word embeddings.
- For dp the embeddings layer outputs weights trained in a dense neural net layer. These will be the word embeddings.

The running time for each algorithm in these areas was measured for comparison.

PART II: Efficiency Theory

2.1. Non-Neural Net Approach (sk and st Algorithms)

2.1.1. Embedding Algorithm

Input: text data

Output: matrix of word embeddings of vocabulary items from text data

get sparse matrix of tf-idf values using vectorizers on text data input

do Singular Value Decomposition (SVD) of sparse matrix of tf-idf values

return matrix of word embeddings

Vectorizers

(a) sk uses sklearn.feature_extraction.text.TfidfVectorizer, which converts a corpus of text documents into a document-term matrix of tf-idf values.

See https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

(a) st uses tf.keras.layers.TextVectorization, which is a preprocessing layer that maps words (strings) to either integer or float (tf-idf) values. It outputs tf-idf values for st (while it outputs integer values for dp). It returns a 1D tensor of values per word (string). The result is a document-term matrix of tf-idf values.

See https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization

SVD

(a) sk uses sklearn.decomposition.TruncatedSVD, which reduces the dimensionality of the tf-idf matrix passed on by the TfidfVectorizer by performing Truncated SVD (see section 1.3 below). The attribute **components_** returns a component-term matrix where the dimension of component is the truncated size, and the dimension of term is the number of terms. As both st and dp return term-component matrices, the matrix returned by this method is transposed to conform.

See https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

(b) st uses tf.linalg.svd, which computes either full SVD or Thin SVD (default) of one or more matrices (see section 1.3 below). Among others, it returns a term-component matrix.

See https://www.tensorflow.org/api_docs/python/tf/linalg/svd

2.1.2. tf-idf Values (Word Vectorization)

Vectorizers transform a dataset of text documents into a matrix where the columns represent terms (words) and the rows represent the documents or texts they appear in. The components of this matrix are tf-idf values. The tf stands for "term frequency" and the idf for "inverse document frequency". The term frequency (tf) of a word w in a document d is the number of times w appears in d divided by the number of words in d. This division occurs to normalize the value, i.e., to put it between 0 (w does not occur in d) and 1 (d consists of just instances of w), inclusive. Thus, tf is given by

tf(w,d) = the number of w in d / the number of words in d

Now, the document frequency (df) of a word is the number of documents in which the word under consideration appears at least once. To normalize this value, it is divided by N, the number of documents in the corpus. This df ratio is very high for common words without meaningful content called stop words, which include "and", "the", "am", "of", etc. (For a list see Bleier, 2010). To reduce their influence, we take the inverse of the document frequency (the idf), which is given by idf(w) = N/df. However, when we have a large corpus, N is very large, so the logarithm is taken. Also, a 1 is added to the denominator in case df is zero. This gives idf as:

idf(w, N) = log(N/(df + 1)).

As a result, the tf-idf value for a word w in a document d is the product of its two components:

tf-idf(w,d, N) = tf(w,d) * log(N/(df + 1)).

See Scott (2019), Wikipedia (tf-idf).

Each word can then be represented by the tf-idf values in the corresponding rows; this is the vector representation of the word in the semantic space determined by the matrix. However, such a matrix is very large and sparse with mostly zeros (Kintsch 2001:176). To deal with this matrix is very costly, so a dense matrix reduction can be found using Singular Value Decomposition (SVD). The vectorizers for sk and st produce tf-idf document-term (document-vocabulary) matrices that are very sparse (mostly full of zeros and with few non-zero elements) and very large. If M is an mxn document-term matrix, with m = the number of documents and n = the number of terms, it would be very large for large m and n. In Natural Language Processing, text corpora tend to have millions to billions of vocabulary items (English-Corpora.org, n.d.) For this paper there are only 20000 documents involved and up to about 55000 vocabulary items. The sk and dp vectorizers produce matrices of shape 20000 x 54621, and 20000 x 54653, respectively, as there are 20000 documents and sk gets 54621 vocabulary items from the data while dp gets 54653. However, the vectorizer for st produces a matrix of shape 19930 x 20000.

A smaller matrix of embeddings is needed, and this is where SVD comes in.

2.1.3. SVD - Singular Value Decomposition

It is applied to reduce the dimensionality of the document-term sparse matrix in order to speed up processing time. In SVD, the tf-idf document-term matrix M is decomposed into a product of three matrices: $U*\Sigma*V^T$ (where V^T is the transpose of V – its rows and columns are interchanged). If M is an mxn matrix, with m = the number of documents and n = the number of terms, then the three components have the following dimensions: U is mxm, Σ is mxn, and V^T is nxn (Wikipedia, Singular value decomposition). Σ is a diagonal matrix, i.e., its non-zero values can only be found on its diagonal. The values along the diagonal (which can include zeros as well) are called the singular values of M. These values indicate "the most important semantic information in the text" (Wikipedia, Latent semantic analysis). As only V^T has the terms, it is used as the new matrix of embeddings, which is a lot smaller and denser (with fewer non-zero elements) than the original in the case that there are more documents than terms. However, full SVD is expensive to compute, so reduced versions are used instead. There are three of these: Thin SVD, Compact SVD, and Truncated SVD (Wikipedia, Singular value decomposition). We will only be using the Thin and Truncated versions.

Truncated SVD (used by sk): The tf-idf document-term matrix M is ***approximately*** decomposed into $U_t*\Sigma_t*V_t^T$ as the smallest non-zero singular values of Σ_t are truncated (left out). If M is an mxn matrix, then the three components have the following dimensions: U_t is mxt, Σ_t is txt, and V_t^T is txn. As only the t largest singular values of Σ are used, only the corresponding t column vectors of U and t row vectors of V^T and the corresponding t rows and columns of Σ are used in the calculation (Wikipedia, Singular value decomposition). If the value of t is picked so that it is very small, this greatly reduces computation of the SVD as most of the rows and columns of the original U, Σ, and V^T are left out. The value of t is typically between 100 and 300 dimensions for NLP purposes (Wikipedia, Latent semantic analysis); in the case of this paper, it is only 50 for laptop computation limitations.

Thin SVD (used by st): The tf-idf document-term matrix M is decomposed into $U_k*\Sigma_k*V_k^T$. If M is an mxn matrix, then the three components have the following dimensions: U_k is mxk, Σ_k is kxk, and V_k^T is kxn. Only the first k column vectors of U and the corresponding first k row vectors of V^T and the corresponding first k rows and columns of Σ are used in the calculation. However, k = min{m, n} (k is the minimum of m and n) and is ***not*** chosen (Wikipedia, Singular value decomposition). Thus, the value of k is very large for a large number of documents and a large vocabulary size, which makes Thin SVD hard to compute in such cases. Compare this to Truncated SVD, which can deal with the much smaller matrices resulting from a small value of t. However, it remains to be seen if Truncated SVD also fails for large t. Note: to make st output an embedding matrix of the same shape (number_of_terms, t) as those produced by sk and dp, the V^T outputted by st's Thin SVD method was truncated to the first t = 50 columns manually.

2.2. Neural Net Approach (dp Algorithm)

This is based on an online tutorial for word embeddings from TensorFlow (TensorFlow.org, 2022).

Embedding Model

A model with the following layers is used (TensorFlow.org, 2022):

6. TextVectorization Layer – input: strings; output: integer values for the vocabulary items of the string input. Shape: (batch, sequence_length)
7. Embedding Layer – input: sequences of vocabulary integer indices; output: dense floating point embedding vectors of variable length. Shape: (batch, sequence_length, embeddings)
8. GlobalAveragePooling1D Layer – input: embedding vectors of variable length; output: embedding vectors of fixed length
9. Dense Layer (neural net layer), in this case of 50 nodes (= truncated value of st and sk)
10. Output layer: Dense Layer of one node.

TextVectorization Layer: transforms strings (one string consisting of an entire document from the corpus) into integer values, one for each vocabulary item of the string, and then feeds these values into the Embedding Layer (TensorFlow.org, 2022b).

Embedding Layer: acts like a lookup table that maps integer indices (standing for words) to their dense floating point vector embeddings, so that if you pass the layer an integer, it will replace it with a floating-point vector from the embedding table. As with any layer, its weights are randomly initialized and are adjusted during training by backpropagation. Once trained, the embeddings will encode similarities between words based on the specific data the model was trained on. The dimensionality of the layer is a parameter that can be experimented with to see what works best just like the number of nodes in a Dense Layer. It outputs a 3D tensor of shape (batch, sequence_length, embedding) (TensorFlow.org, 2022). An Embedding Layer is used instead of a Dense Layer as the Embedding Layer reads results right off the weights matrix, whereas a Dense Layer needs to do matrix multiplications to reach the same results. So, the Embedding Layer is faster (Stack Overflow, 2019).

GlobalAveragePooling1D Layer: this returns a fixed-length vector for each vector outputted by the Embedding Layer by averaging over the sequence dimension. RNN or Attention layers could have been used, but pooling is the simplest (TensorFlow.org, 2022).

Embedding Dimensions: The deep learning algorithm, dp, depends on the number of embedding dimensions, which will equal the length of the word vectors produced by the algorithm. The higher the number of embedding dimensions, the longer it takes to calculate the embedding weights that will end up being the word embeddings. For small values, the algorithm will perform fairly quickly, depending on the number of epochs it needs to run to train the model. It has been found for this paper (using only 20000 documents) that ten epochs are required to get an accuracy of around 80%. To fairly compare sk, st, and dp, the number of embedding dimensions is set to the value of t used by sk, which is also the number of columns left after manual truncation in st.

Output: To get the embeddings matrix model.get_layer('embedding').get_weights()[0] is used. It returns a matrix of shape term x component (Krogager, 2019).

PART III: Efficiency Results

3.1. Preliminary Results on the Running Times of Three Word-Embedding Algorithms

3.1.1. Tabular Data

The below table indicates the running times in seconds for each algorithm for a given number of vocabulary items to be processed. There are three trial runs for each vocabulary size to take into account fluctuations in performance.

TABLE 1: DATA up to vocabulary size = 3500 (out of a total of 20000), truncation = 50

vocab	trial	sk	st	dp
500	1	2.835821	12.32562	31.48812
500	2	2.739016	13.26875	26.89305
500	3	2.747003	13.04799	26.70584
1000	1	2.847053	24.45132	25.55259
1000	2	3.282994	22.25137	24.73161
1000	3	2.840024	22.71621	25.52694
1500	1	2.82906	41.02082	25.53761
1500	2	2.928027	40.1609	26.20766
1500	3	2.843997	41.45011	24.80209
2000	1	3.149027	64.81428	25.54459
2000	2	3.174069	63.83087	25.63846
2000	3	2.873999	64.47922	26.56449
2500	1	2.919001	95.67032	26.35444
2500	2	2.959022	95.36113	25.75291
2500	3	2.918001	94.74267	25.66315
3000	1	3.453998	130.8365	25.60226
3000	2	2.943075	132.8487	25.44353
3000	3	2.987017	132.3833	26.49861
3500	1	2.998998	175.9252	26.36553
3500	2	3.087999	174.5949	25.96059
3500	3	3.472996	174.9189	27.31386

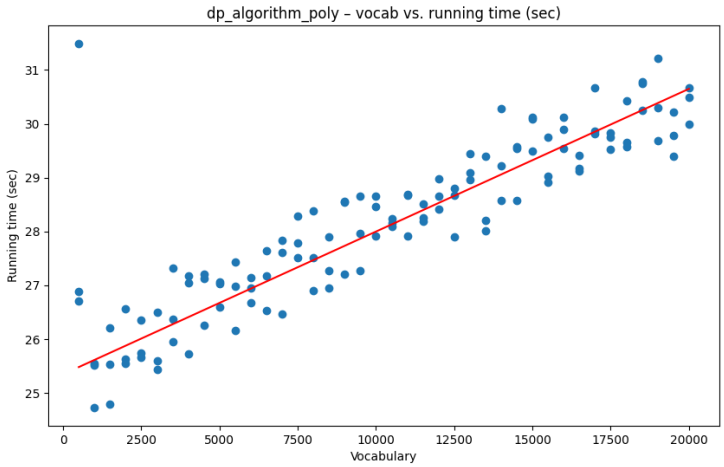
*sk breaks down in a for-loop writing out to a csv file at vocabulary size 3000, truncation size 29. However, there are no limitations when used in a for-loop printing out to console. In contrast, there do not seem to be such restrictions for st and dp. Especially st as it calculates the whole Thin SVD and then truncates it.

**st only goes up to vocabulary size 4500 due to the cost of the calculations.

3.1.2. Graphs

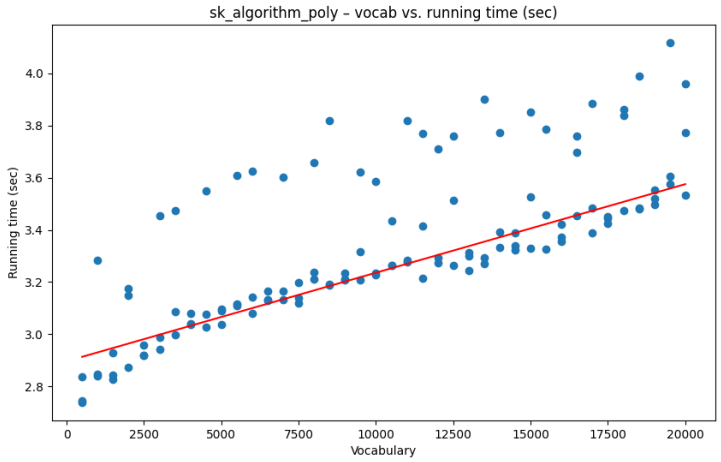
The below graphs show the results of linear regression for all algorithms. Random Sample Consensus (RANSAC) regression was used for all of them to estimate the final models using only the inliers (Theil-Sen regression could also have been used) (Lewinson, 2022). The graphs for dp and sk are accompanied by the corresponding R^2 values and the coefficients of the line of best fit.

Linear dp



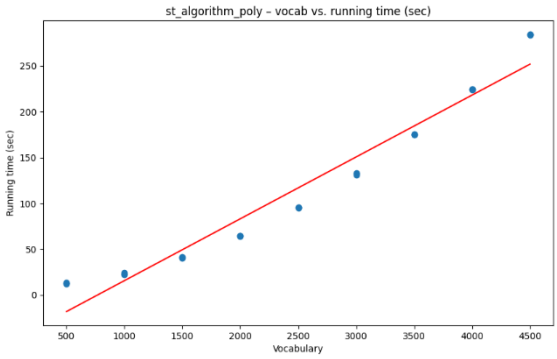
- R^2 = 0.7583745883882209
- Coefficients = [25.351261478069603, 0.00026484]

Linear sk

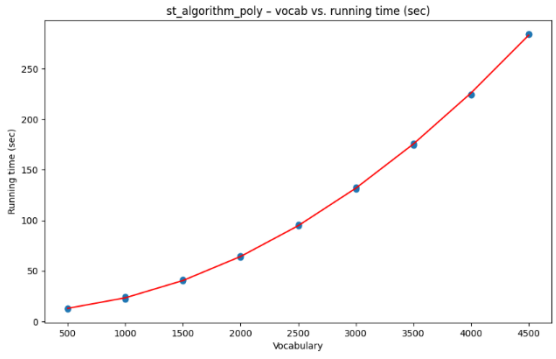


- R^2 = 0.43607755051082897
- Coefficients = [2.8969173854743193, 3.39105851e-05]

st is NOT Linear but (Probably Quadratic) Polynomial



LINEAR ATTEMPT



QUADRATIC

Evidently, st is not linear but polynomial, most likely quadratic. This is because st uses tf.linalg.svd, which is Thin SVD. It is known that the Full SVD time complexity for an mxn matrix is $O(\min(mn^2, m^2n))$, while for Truncated SVD (used by sk) it is $O(k^2m)$, which for a given constant k is O(m) (Math Overflow, 2019).

3.1.3. R^2 Values

Even though R^2 values do not indicate very much (Ford, 2015; Frost, n.d., Penn State, 2018), Table 2 below provides R^2 values according to polynomial (with degree indicated), logarithmic, and exponential models, and the values do not differ by much from one model to another (with the exception of log), so the simplest model is chosen for sk and dp: the degree 1 polynomial.

TABLE 2: R^2 values for all algorithms according to polynomial (with degree), log, and exp models

Degree	dp	sk	st
1	0.7583745883882209	0.43607755051082897	0.9522517176053572
2	0.7509106662905711	0.41236036583702185	0.9998976034137815
3	0.7702471312229546	0.42227322044586824	0.9999150944253051
4	0.7350859115347191	0.43512600016619307	0.9999457228110203
log	0.5193793347963686	0.3833408431752485	0.576412849586986
exp	0.7591482222769029	0.46111658251933685	0.9711141717695009

3.1.4. Relative Growth Rate

As the linear model was chosen for sk and dp, the algorithms' growth rates can be compared by dividing one of their slopes by the other. Let the ratio be that of greater slope to lesser slope. As a result, the relative growth rate can be estimated as follows:

$$\text{slope dp / slope sk} = 0.00026484 / 3.39105851\text{e-}05 = 7.80995 \approx 7.8$$

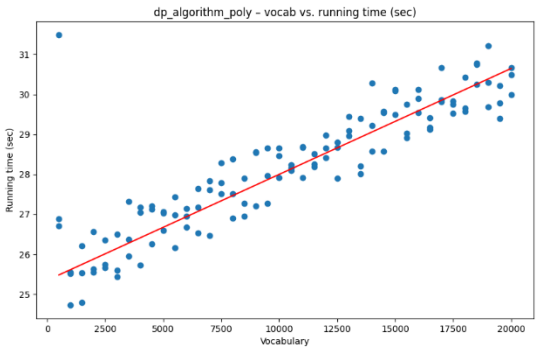
Thus,

- dp grows about 7.8 times faster than sk, and sk is thus faster in execution than dp.
- In terms of boundedness, sk is bounded above by dp, which is bounded above by st. So, sk is the fastest model in terms of execution.

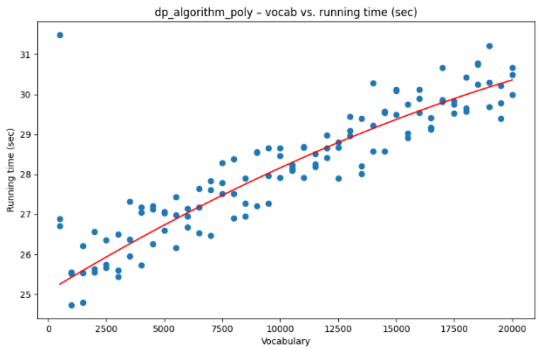
3.1.5. Polynomial Regression Graphs for the Algorithms

From left to right, top to bottom: linear, quadratic, cubic, and quartic.

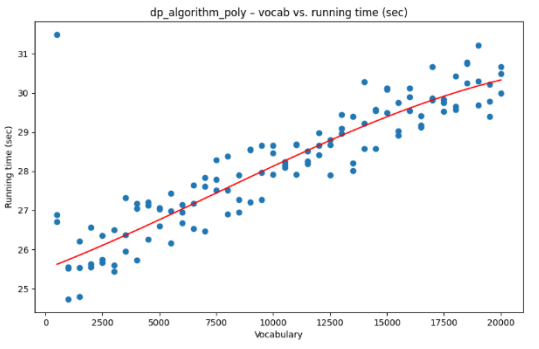
dp graphs:



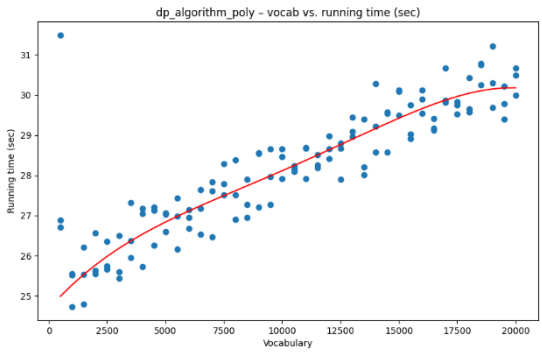
LINEAR



QUADRATIC

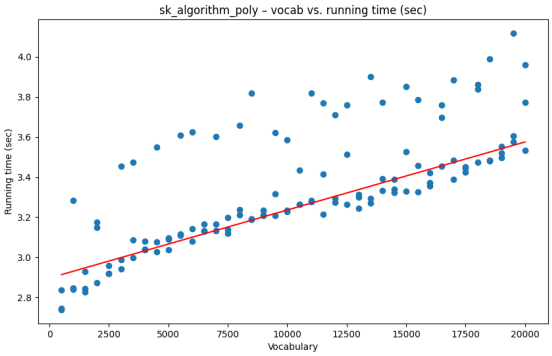


CUBIC

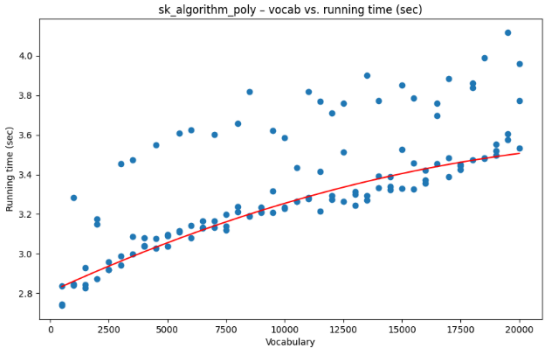


QUARTIC

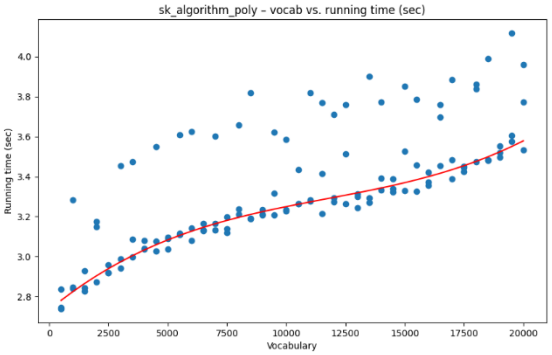
sk graphs:



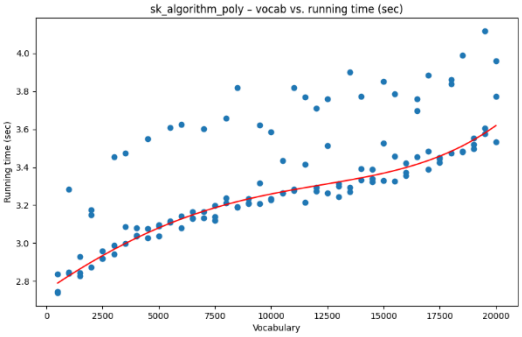
LINEAR



QUADRATIC

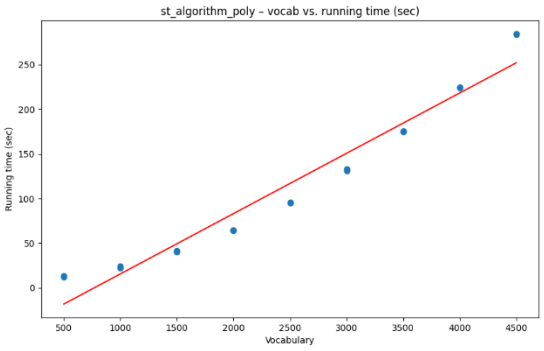


CUBIC

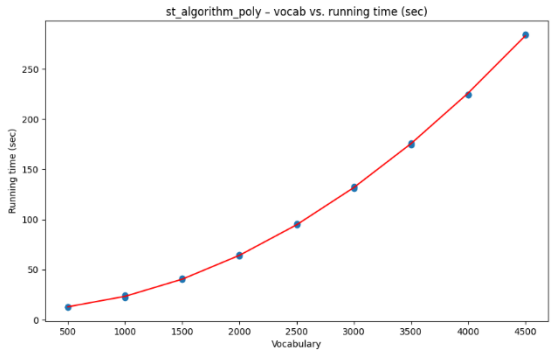


QUARTIC

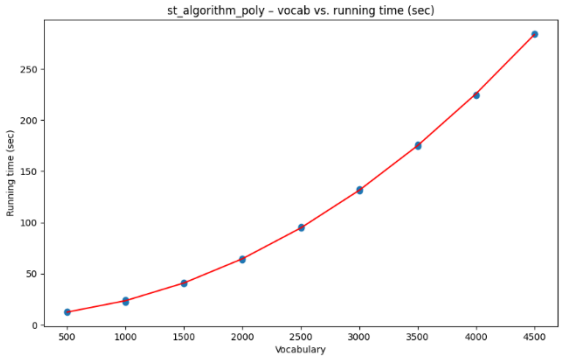
st graphs:



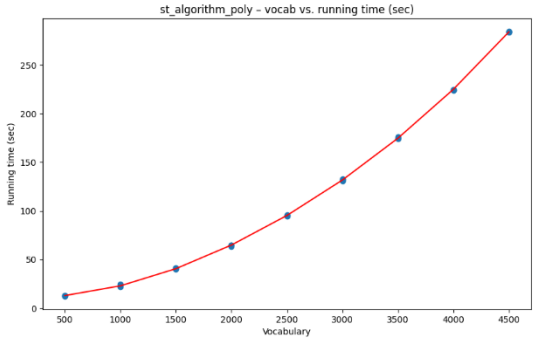
LINEAR



QUADRATIC



CUBIC



QUARTIC

3.2. Conclusions on Running Times

- **Relative growth rate:** dp grows about 7.8 times as fast as sk, and st is likely quadratic. As a result, sk runs the fastest in terms of execution.

- **Time complexity:** sk and dp are linear with O(n), whereas st is likely quadratic O(n^2). This is probably due to st using tf.linalg.svd, which uses a Thin SVD method.

PART IV: Effectiveness Theory

4.1. Effectiveness of the Algorithms: Cosine Similarities between Words, Dot Products, and Kernelization Methods

Cosine Similarity as a Measure of Algorithm Effectiveness: So far, the algorithms have been compared with respect to their efficiency, but there is the question of their effectiveness. How are they to be compared? The effectiveness of an embedding has to do with whether or not it can show that two related word vectors are close together in the semantic vector space. This distance is measured by the cosine of the angle between them.

Two words can be seen as related if their vectors are separated by a small angle. If the two words are the same, then this angle is zero. If they are completely unrelated, this angle is 90^o, and if completely opposite in meaning, the angle is 180^o. The corresponding cosine values for these angles are 1, 0, and -1. So, the cosine between two word vectors is a measurement of the relatedness of their meanings. Cosines can be calculated using the dot products of the vectors involved, say x and z, according to the formula:

$\cos(\theta_{x,z}) = < x, z> / (||x|| \bullet ||z||)$.

Where $< x, z>$ is the dot product of x and z and $||x||$ is the norm, or magnitude of x (its length). There is another simplification: the norm of a vector is defined as the square root of its dot product with itself, so we get

$\cos(\theta_{x,z}) = < x, z > / (< x, x >^{\frac{1}{2}} \bullet < z, z >^{\frac{1}{2}})$

Neighboring Words as a Subjective Measure of Effectiveness: One very useful qualitative measure of the effectiveness of the algorithms is to see what word vectors are nearest to a given word vector. If many of these neighboring words are related, then the corresponding algorithm can be said to be effective. However, this type of estimation is rather subjective in nature.

Kernel Version of the Cosine between Two Word Vectors: The standard dot product between two column vectors x and z is just x^Tz, where x^T is x transpose, which converts the column vector x into a row vector as two column vectors cannot be multiplied together, but only a row vector with a column vector.

If we take two column vectors with d features each, there are two corresponding vectors that contain these same features and all of their possible combinations (Weinberger, 2017, 2018; Wikipedia, Polynomial kernel) for a total of 2^d polynomial features each (Weinberger, 2017, 2018). It can be hypothesized that with more dimensions, a word embedding will be more precise in establishing word meaning. However, the dot product of such high-dimensional vectors is computationally intractible given the number of components involved, but there is a polynomial kernel equivalent that is quick to compute – this is known as the "kernel trick" (Weinberger, 2017, 2018; Wikipedia, Polynomial kernel):

$k(x, z) = (x^Tz + 1)^d$.

So, Replacing the dot product by its kernel equivalent gives the following calculation for the cosine between to word vectors:

$\cos(\theta_{x,z}) = k(x,z) \,/\, (k(x,x)^{\frac{1}{2}} \bullet k(z,z)^{\frac{1}{2}})$

Both standard and kernel versions will be used in this report.

4.2. Cosine Algorithms - Standard and Kernel Versions

Cosine Similarity Algorithm (Standard / Kernel-Based)

Input: Two strings (words)

Output: Cosine, between the two word vectors corresponding to the two strings, based on the standard dot product or n-polynomial kernel dot product

Time Complexity: O(n)

get the standard dot product or n-polynomial kernel dot product of the two vectors

get the standard dot product or n-polynomial kernel dot product of one of the vectors with itself **and** take the square root

get the standard dot product or n-polynomial kernel dot product of the other vector with itself **and** take the square root

return the division of the first value by those of the second and third

n-Closest Algorithm (Standard / Kernel-Based)

Input: a string (word)

Output: a dictionary of the top n closest vocabulary items (strings) to the input (included) along with corresponding similarity values

get dictionary of vocabulary items as keys and their cosine similarities (either standard or kernel-based) to the input as values

reverse sort the dictionary

return a dictionary of the first n key-value pairs from the sorted dictionary

PART V: Effectiveness Results

5.1. Results for Cosine Calculations - Standard and Kernel Versions - for a Vocabulary Size of 20000 (4500 for st) and a Truncation Size of 50

5.1.1. dp Results

Cosine Similarity Results for "audience" and "mission":

- Standard: 0.5056709051132202
- 2-polynomial kernel: 0.8555729389190674
- 3-polynomial kernel: 0.7913809418678284

Comment: There is a dissimilarity between the standard and kernel cosines.

Cosine similarity between "audience" and "audience" to see if the result is 1.0 (expected):

- Standard: 1.0000001192092896
- 2-polynomial kernel: 1.0
- 3-polynomial kernel: 0.9999998807907104

Comment: Tests passed.

10 closest vocabulary items for "audience" with corresponding cosine similarities:

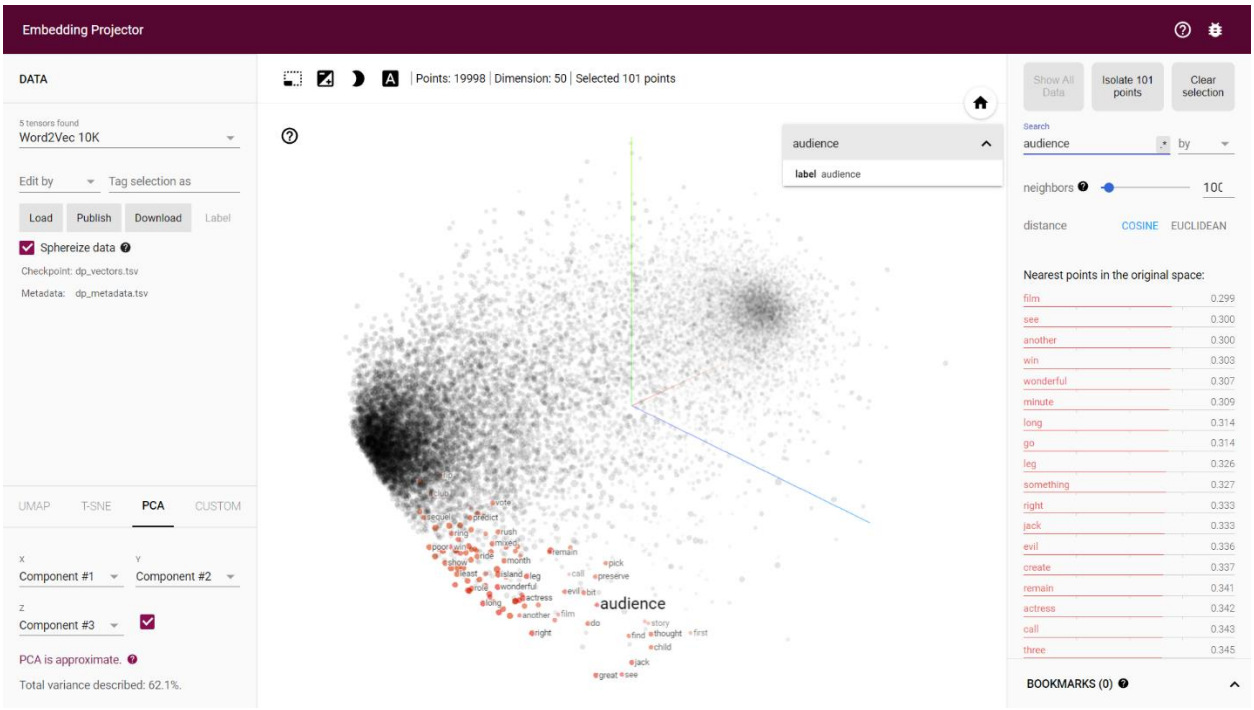
Using normal cosine similarity: [('audience', 1.0000001), ('film', 0.7228071), ('see', 0.72091156), ('go', 0.715287), ('another', 0.71378535), ('win', 0.7043159), ('wonderful', 0.6945388), ('minute', 0.69374585), ('leg', 0.6878335), ('jack', 0.6876035)]

Using 2-polynomial kernel cosine similarity: [('audience', 1.0), ('remain', 0.94936633), ('leg', 0.94579697), ('evil', 0.94524026), ('pick', 0.9443906), ('seek', 0.9437649), ('bit', 0.9436575), ('call', 0.94171524), ('preserve', 0.94076514), ('jack', 0.94011265)]

Using 3-polynomial kernel cosine similarity: [('audience', 0.9999999), ('remain', 0.92501915), ('leg', 0.91980726), ('evil', 0.9189953), ('pick', 0.9177564), ('seek', 0.91684455), ('bit', 0.9166881), ('call', 0.9138595), ('preserve', 0.9124768), ('jack', 0.91152763)]

Comment: The words bear little similarity to “audience”, except for perhaps “film” and “see” in the first group.

Embeddings Projector for dp



Word embeddings from dp using the Embeddings Projector at https://projector.tensorflow.org/ for 19998 vocabulary items (two special tokens were excluded) represented by 50-element vectors.

5.1.2. sk Results

Cosine Similarity Results for "audience" and "mission":

- Standard: 0.0031140797479220776
- 2-polynomial kernel: 0.9833302526450598
- 3-polynomial kernel: 0.975099875483772

Comment: There is a great dissimilarity between the standard and kernel cosines.

Cosine similarity between "audience" and "audience" to see if the result is 1.0 (expected):

- Standard: 1.0000000000000002
- 2-polynomial kernel: 1.0
- 3-polynomial kernel: 0.9999999999999998

Comment: Tests passed.

10 closest vocabulary items for "audience" with corresponding cosine similarities:

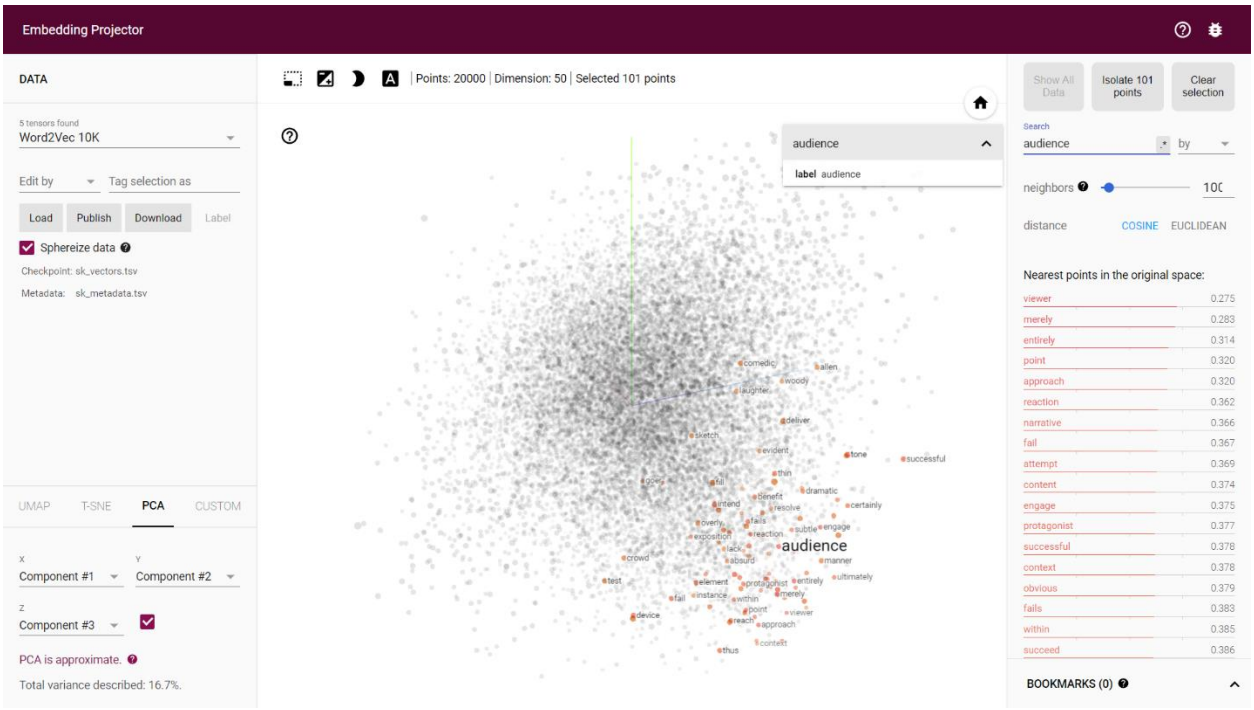
Using normal cosine similarity: [('audience', 1.0000000000000002), ('viewer', 0.6707602346878841), ('merely', 0.6585099779672122), ('entirely', 0.6270793708314674), ('approach', 0.6202657899665506), ('point', 0.6137397859732775), ('overt', 0.5882510849015894), ('reaction', 0.58711353900759), ('goer', 0.5827367015203715), ('content', 0.5737913227600511)]

Using 2-polynomial kernel cosine similarity: [('audience', 1.0), ('viewer', 0.9911267038359861), ('fail', 0.9897405179830602), ('element', 0.9895019134541243), ('fails', 0.9893995962203415), ('within', 0.9893768046399499), ('obvious', 0.9893311079835722), ('perhaps', 0.9892239133927633), ('approach', 0.9891879333239783), ('meant', 0.989117355940002)]

Using 3-polynomial kernel cosine similarity: 10 closest vocabulary items for "audience" using 3-polynomial kernel cosine similarity: [('audience', 0.9999999999999998), ('viewer', 0.9867196253343768), ('fail', 0.9846503160924414), ('element', 0.9842942714623033), ('fails', 0.9841416072851449), ('within', 0.9841076017931216), ('obvious', 0.9840394226513066), ('perhaps', 0.9838794951330418), ('approach', 0.9838258170988726), ('meant', 0.9837205267719834)]

Comment: The words bear little similarity to “audience” with the exception of “viewer”.

Embeddings Projector for sk



Word embeddings from sk using the Embeddings Projector at https://projector.tensorflow.org/ for 20000 vocabulary items represented by 50-element vectors.

5.1.3. st Results

Cosine Similarity Results for "audience" and "mission":

- Standard: -0.31769856810569763
- 2-polynomial kernel: 0.9385449886322021
- 3-polynomial kernel: 0.9092485308647156

Comment: There is a great dissimilarity between the standard and kernel cosines.

Cosine similarity between "audience" and "audience" to see if the result is 1.0 (expected):

- Standard: 1.0
- 2-polynomial kernel: 1.0
- 3-polynomial kernel: 0.9999998807907104

Comment: Tests passed.

10 closest vocabulary items for "audience" with corresponding cosine similarities:

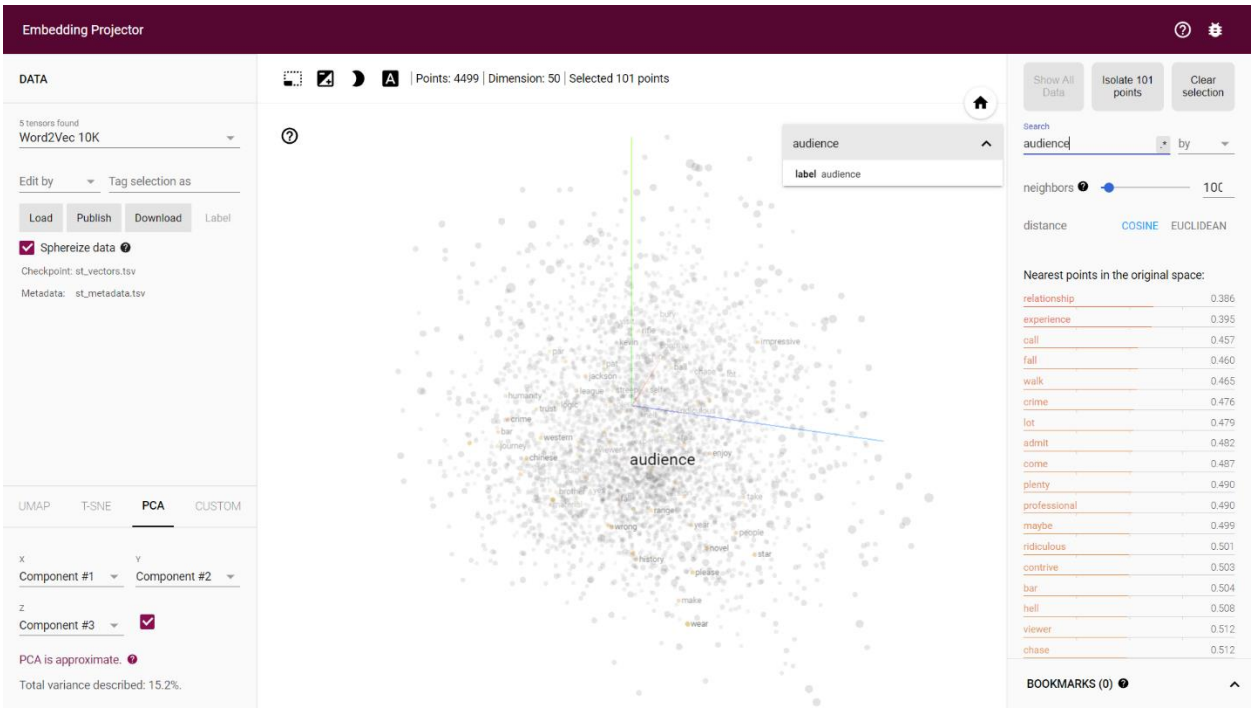
Using normal cosine similarity: [('audience', 1.0), ('relationship', 0.61856747), ('experience', 0.609516), ('professional', 0.5495951), ('jackson', 0.54911625), ('contrive', 0.5469533), ('walk', 0.5425551), ('call', 0.5422834), ('league', 0.5388335), ('fall', 0.53409714)]

Using 2-polynomial kernel cosine similarity: [('audience', 1.0), ('relationship', 0.9935875), ('experience', 0.99352586), ('walk', 0.9926242), ('plenty', 0.992564), ('ridiculous', 0.99242836), ('fall', 0.9923885), ('admit', 0.9923846), ('bar', 0.9923834), ('hell', 0.99230623)]

Using 3-polynomial kernel cosine similarity: [('audience', 0.9999999), ('relationship', 0.9903967), ('experience', 0.99030447), ('walk', 0.9889568), ('plenty', 0.9888668), ('ridiculous', 0.98866415), ('fall', 0.9886045), ('admit', 0.98859864), ('bar', 0.988597), ('hell', 0.9884815)]

Comment: The words bear little similarity to “audience” except for “experience”, like experiencing an event.

Embeddings Projector for st



Word embeddings from st using the Embeddings Projector at https://projector.tensorflow.org/ for 4999 vocabulary items (one special token was excluded) represented by 50-element vectors.

5.2. Cosine and Kernel Conclusions

- There is a great dissimilarity between the standard and kernel cosines for all algorithms.
- All algorithms passed the tests requiring that the cosine of a word vector with itself be 1.0 (minor discrepancies are probably due to limitations on representing decimals with floating points).
- The 10-closest words bore little resemblance to the target word "audience" except for two relevant words from dp and one relevant word each from sk and st. The explanation for this is that the trunc_size is only 50, whereas the size needed for embeddings to mimic human semantics is from 100 to 300 (Wikipedia, Latent semantic analysis). However, such values are rather difficult to process on a laptop.

References

Bleier, S. 2010. NLTK's list of english stopwords. https://gist.github.com/sebleier/554280

Dunn, K. 2021. Avoid R-squared to judge regression model performance. https://towardsdatascience.com/avoid-r-squared-to-judge-regression-model-performance-5c2bc53c8e2e

English-Corpora.org. n.d. Size. https://www.english-corpora.org/size.asp#:~:text=Corpus%20size%20is%20incredibly%20important,400%20times%20as%20much%20data.

Ford, C. 2015. Is R-squared Useless? https://data.library.virginia.edu/is-r-squared-useless/

Frost, J. n.d. How To Interpret R-squared in Regression Analysis. https://statisticsbyjim.com/regression/interpret-r-squared-regression/

Kintsch, W. 2001. Predication. *Cognitive Science* 25:173-202.

Krogager, A. 2019. keras embedding weights lookup with categorical variables (answer). https://stackoverflow.com/questions/54430475/keras-embedding-weights-lookup-with-categorical-variables

Lewinson, E. 2022. Dealing with Outliers Using Three Robust Linear Regression Models. https://developer.nvidia.com/blog/dealing-with-outliers-using-three-robust-linear-regression-models/

Martin, Dian I., and Michael W. Berry. 2007. Mathematical Foundations behind Latent Semantic Analysis. In *Handbook of Latent Semantic Analysis*, ed. T. K. Landauer, Mahwah, New Jersey: Lawrence Erlbaum Associates.

Math Overflow. 2019. What is the time complexity of truncated SVD? https://mathoverflow.net/questions/161252/what-is-the-time-complexity-of-truncated-svd/221216#221216

Pennsylvania State University. 2018. 2.8 - R-squared Cautions. https://online.stat.psu.edu/stat462/node/98/

Scott, W. 2019. TF-IDF from scratch in python on a real-world dataset. https://towardsdatascience.com/tf-idf-for-document-ranking-from-scratch-in-python-on-real-world-dataset-796d339a4089#:~:text=TF%2DIDF%20stands%20for%20%E2%80%9CTerm,Information%20Retrieval%20and%20Text%20Mining.

Stack Overflow.2019. What is the difference between an Embedding Layer and a Dense Layer? https://stackoverflow.com/questions/47868265/what-is-the-difference-between-an-embedding-layer-and-a-dense-layer

TensorFlow.org. 2022. Word embeddings. https://www.tensorflow.org/text/guide/word_embeddings

TensorFlow.org. 2022b. tf.keras.layers.TextVectorization. https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization

Weinberger, K. 2018. Lecture 13: Kernels. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote13.html

Weinberger, K. 2017. Machine Learning Lecture 22 "More on Kernels" -Cornell CS4780 SP17 https://www.youtube.com/watch?v=FgTQG2IozlM

Wikipedia, Latent semantic analysis. https://en.wikipedia.org/wiki/Latent_semantic_analysis#Rank-reduced_singular_value_decomposition

Wikipedia. Polynomial kernel. https://en.wikipedia.org/wiki/Polynomial_kernel

Wikipedia. Singular value decomposition. (Last modified on 30 March 2016, at 12:06.) https://en.wikipedia.org/wiki/Singular_value_decomposition#Reduced_SVDs

Wikipedia. tf-idf. https://en.wikipedia.org/wiki/Tf%E2%80%93idf

Wikipedia. Word embedding. https://en.wikipedia.org/wiki/Word_embedding