Language Recogniton using Random Indexing

Implimentation in C language

Varun Rama Raju Dandu
Graduate School of Engineering
San Francisco State University
San Francisco, CA 94132, USA.
vdandu@mail.sfsu.edu

Kautilya Kukunoor
Graduate School of Engineering
San Francisco State University
San Francisco, CA 94132, USA.
kkukunoo@mail.sfsu.edu

***Abstract*—Random Indexing is a simple implementation of Random Projections with a wide range of applications. It solves many problems without much complexity. Here we demonstrate its use for identifying the language of text samples, by encoding n-gram into high-dimensional Language Vectors. We will show that the method is easily implemented and required little computational power and space. We implement this method using C-Language and will show you the success in this language-recognition task. On a difficult data set of 21,000 short sentences from 21 different languages, we achieve 97.8%(D = 15,000 & N = 4) accuracy.**

I. INTRODUCTION

We humans communicate through language, we have an extraordinary ability to recognize the unknown languages in spoken or written form, using simpel cues to distinguish one language from another. Some familiar languages, ofcourse from the same language family might sound very similar. This is because embedded in every language are certain features the at clearly distinguish from one and other. The same can be sad with written languages. First steps of all sorts for language processong is recognizing the language itself, such as text analysis, categorizationa, translation etc.

Popularized by shannon (1948).

The theory says, most language model use distributional statistics to explain structural similarities in various specified languages. The traditional method of identifying languages consists of counting individual letters, letter bigrams, trigrams, tetra-grams, etc., and comparing the frequency profiles of different text samples. As a general principle, the more accurate you want your detection method to be, the more data you have to store about the various languages. For example, Google's recently open-sourced program called Chromium Compact Language Detector uses large language profiles built from enormous corpora of data. As a result, the accuracy of their detection, as seen through large-scale testing and in practice, is near perfect (McCandless (2011))

High-dimensional vector models are popular in natural language processing and are used to capture word meaning from word-use statistics. The vectors are called semantic vectors or context vectors.

Ideally, words with a similar meaning are represented by semantic vectors that are close to each other in the vector space, while dissimilar meanings are represented by semantic vectors far from each other.Hyper-dimensions based on Random Projections, was proposed by Papadimitriou et al. and Kaski (1998). Random Indexing (Kanerva et al. (2000); Sahlgren (2005)) is a simple and effective implementation of the idea. It has been used in ways similar to Mikolov et al.'s Continuous Bag of Words Model (KBOW; Mikolov & Dean (2013)) and has features similar to Locality-Sensitive Hashing (LSH) but differs from them in its use of high dimensionality and randomness. With the dimensionality in the thousands (e.g., D = 10,000)—

The idea is based on a paper "Language Recognition using Random Indexing" by Joshi, Halseth and Kanerva.

referred to as “hyper-dimensional”—it is possible to calculate useful representations in a single pass over the dataset with very little computing.
In this paper, we will present a way of doing language detection using Random Indexing, which is fast, highly scalable, and space efficient. We will also present some results regarding the accuracy of the method, even though this will not be the main goal of this paper and should be investigated further.

II. RANDOM INDEXING.

Random Indexing stores information by projecting data onto vectors in a hyperdimensional space. There exist a huge number of different, nearly orthogonal vectors in such a space (Kanerva, 1988, p. 19). This lets us combine two such vectors into a new vector using well-defined vector-space operations, while keeping the information of the two with high probability. In our implementation of Random Indexing, we use a variant of the MAP (Multiply, Add, Permute) coding described in Levy & Gayler (2009) to define the hyperdimensional vector space. Vectors are initially taken from a D -dimensional space (with D = 10,000) and have an equal number of randomly placed 1s and -1 s. Such vectors are used to represent the basic elements of the system, which in our case are the 26 letters of the alphabet and the (ASCII) Space. These vectors for letters are sometimes referred to as their Random Labels.
The binary operations on such vectors are defined as follows. Elementwise addition of two vectors A and B, is denoted by A + B. Similar, elementwise multiplication is denoted by A * B. A vector A will be its own multiplicative inverse, A * A = 1, where 1 is the D-dimensional identity vector consisting of only 1 s. Cosine angles are used to measure the similarity of two vectors. It is defined as cos(A,B) = |A’ * B’| , where A’ and B’ are the normalized vectors of A and B , respectively, and |C| denotes the sum of the elements in C .
Information from a pair of vectors A and B is stored and utilized in a single vector by exploiting the summation operation. That is, the sum of two separate vectors naturally preserves unique information from each vector because of the mathematical properties of the hyperdimensional space. To see this, note that cos(A*A) = 1, while for all B /= A, cos(A*B) < 1. The cosine of two random, unrelated vectors tend to be close to 0. Because of this, the vector B can easily be found in the vector A + B: cos (B,A + B) differs significantly from 0. For storing sequences of vectors, we use a random (but fixed throughout all our computations) permutation operation ρ of the vector coordinates. Hence, the sequence A-B-C, is stored as thevector (ρ((ρA) ρB)) ρC = ρρA* ρB *C . This efficiently distinguishes the sequence A-B-C from, say, A-C-B. This can be seen from looking at their cosine (here c is the normalization factor):

V1 = ρρA*ρB*C
V2 = ρρA*ρC* B
⇨ cos(V1, V2) = C |(ρρA * ρB * C) * (ρρA*ρC * B)|
= C |ρρA * ρρA *ρB * ρC * C * B)|
= C |ρ ρ (A * A) * ρ (B * C) * (B * C))|
≅ C (0).

since a random permutation ρV1 of a random vector V1 is uncorrelated to V2

2.1 Making and comparing text vectors.

We use the properties of hyperdimensional vectors to extract certain properties of text into a single vector. Kanerva (2014) shows how Random Indexing can be used for representing the contexts in which a word appears in a text, into that word’s context vector. We show here how to use a similar strategy for recognizing a text’s language by creating and comparing Text Vectors: the Text Vector of an unknown text sample is compared for similarity to precomputed Text Vectors of known language samples—the latter are

referred to as Language Vectors.
Simple language recognition can be done by comparing letter frequencies of a given text to known letter frequencies of languages. Given enough text, a text's letter distribution will approach the letter distribution of the language in which the text was written. The phenomenon is called an "ergodic" process in Shannon (1948), as borrowed from similar ideas in physics and thermodynamics. This can be generalized to using letter blocks of different sizes. By a block of size n , we mean n consecutive letters in the text so that a text of length m would have m n+3 blocks. When the letters are taken in the order in which they appear in the text, they are referred to as a sequences (of length n) or as n -grams.
As an example, the text "a brook" gives rise to the tetragrams "–a–b", "a–br", "–bro", "broo", "rook", and "ook–" (here "–" stands for Space). The frequencies of such letter blocks can be found for a text and compared to known frequencies for different languages. For texts in languages using the Latin alphabet of 26 letters (plus Space), like English, this would lead to keeping track of 274 = 531,441 different tetragram frequencies. For arbitrary alphabets of l letters, there would be (l +1) n n -grams to keep track of. These numbers grow quickly as the block size n increases.
The Random Indexing approach for doing language recognition is similar. A text's Text Vector is first calculated by running over all the blocks of size n, within the text and creating an n -Gram Vector for each. An n -Gram Vector is created for the sequence of letters as described earlier. As an example, if we encounter the block "grab", its vector is calculated by performing ρρρG + ρρR + ρA + B ,
where G, R, A and B are the Random Labels for g, r, a, and b—they are random D –dimensional vector with half 1 s and half 1 s and they remain constant.
A text's Text Vector is now obtained from summing the n -Gram Vectors for all the blocks in the text. This is still an D -dimensional vector and can be stored efficiently. Language Vectors are made in exactly the same way, by making Text Vectors from samples of a known language and adding them into a single vector. Determining the language of an unknown text is done by comparing its Text Vector to all the Language Vectors. More precisely, the cosine angle measure dcos between a language vector X and an unknown text vector V is defined as follows:

$$d_{\cos}(X,V) = \frac{X \cdot V}{|X||V|} = \frac{\sum_{i=1}^{D} x_i v_i}{\sqrt{\sum_{j=1}^{D} x_j^2 \sum_{k=1}^{D} v_k^2}}$$

If the cosine angle is high (close to 1), the block frequencies of the text are similar to the block frequencies of that language and thus, the text is likely to be written in the same language. Hence, the language that yields the highest cosine is chosen as the system's prediction/guess.

2.2 Complexity

The outlined algorithm for Text Vector generation can be implemented efficiently. For generating a vector for an n -gram, n - 1 vector additions and permutations are performed. This takes time O (n * D) . Looping over a text of m letters, O(m) n -Gram Vectors must be created and added together. This clearly implies an O (n * D * m) implementation. This can be improved to O (D * m) by noting that most of the information needed for creating the n-Gram Vector for the next block is already contained in the previous n -Gram Vector, and can be retrieved by removing the contribution from the letter that is now no longer in the block. Say we have the n -Gram Vector A = ρ(n 1)V1 * ρ (n 2)V2 * …. * ρVn 1 * Vn for block number i , and now want to find the n -Gram Vector B for block i+1 . We remove from A the vector ρ (n)V1 by multiplying with its inverse (which is the vector itself), which we can do in O(D) time since ρ (n 1) is just another (pre-calculated) permutation. Then we permute the result once using ρ and multiply.

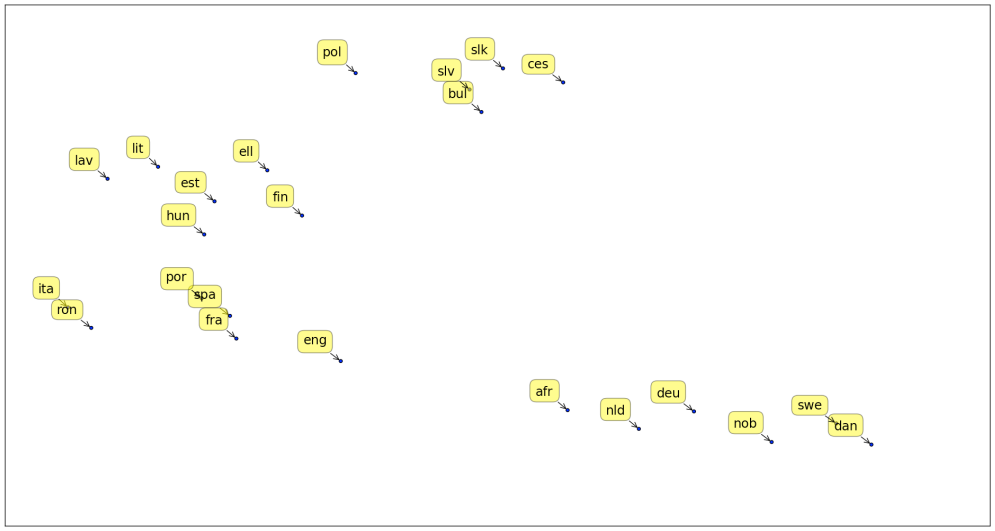


Figure 1: 10,000-dimensional Language Vectors for 23 languages roughly cluster based on the known relations between the languages. The Language Vectors were based on letter trigrams and were projected onto a plane using t-sne (van der Maaten(2008)).

that with the Letter Vector Vn+1 for the new letter in the block. This gives us the new n-Gram Vector

$B = (\rho^{(n-1)}V1 * A) * Vn+1$

$= \rho\,(\rho^{(n-2)}V2 * \ldots * \rho Vn-1 * Vn) * Vn+1$

$= \rho^{(n-1)}V2 * \ldots * \rho^{(2)}Vn-1 * \rho Vn * Vn+1$

and so we can create n-Gram Vectors for arbitrary size blocks without adding complexity. In practice it means that the text is processed in a single pass at about a 100,000 letters a second on a laptop *computer.*

III. EXPERIMENTAL RESULTS

The algorithm outlined above was implemented by (Joshi & Halseth (2014)), and used to create Language Vectors for 23 languages. Texts for the Language Vectors were taken from Project Gutenberg (Hart) where it is available in a number of languages, and from the Wortschatz Corpora (Quastoff et al. (2006)) where large numbers of sentences in selected languages can be easily downloaded. Each Language Vector was based on about 100,000 bytes of text. Computing of the Language Vectors corresponds to training the system and took about 1 second per language on a laptop computer.

Intuitively, Language Vectors within a language family should be closer to each other than vectors for unrelated languages. Indeed, the hyper-dimensional Language Vectors roughly cluster in this manner.

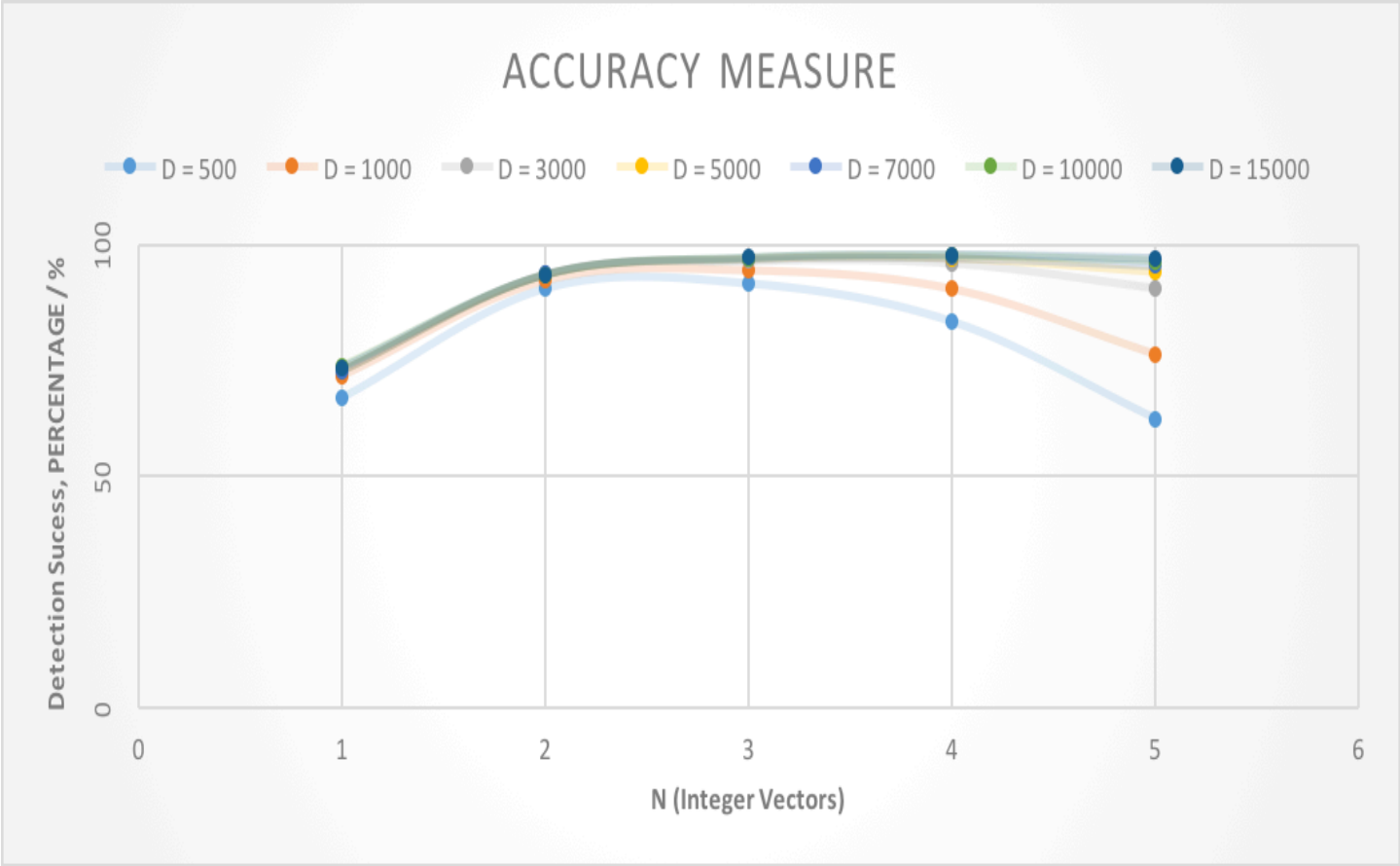
To get an idea of how well the actual detection algorithm works, we tested the Language Vectors' ability to identify text samples from the Europarl Parallel Corpus, described in Nakatani. This corpus includes 21 languages with 1,000 samples of each, and each sample is a single sentence.

Table 1 shows the result for n -gram sizes from 1 to 5 (n = 1 is the equivalent of comparing letter histograms) and also for various sizes of D from 500 to 15,000. With tetragrams we were able to guess the correct language with 97.8% accuracy. Even when incorrect, the system usually chose a language from the same family. For ex: It is worth noting that 10,000-dimensional Language Vectors are keeping track of 531,441 possible tetragrams and easily accommodate pentagrams and beyond if needed.

IV. TABLE AND GRAPH

- Table 1. Detection success table.

ACCURACY IN %	N =1	N = 2	N =3	N = 4	N = 5
D = 500	66.94%	90.671%	91.72%	83.414%	62.338%
D = 1000	71.8%	92.47%	94.609%	90.56%	76.414%
D = 3000	73.05%	93.75%	96.56%	96.08%	90.76%
D = 5000	72.96%	93.54%	96.86%	97.02%	94.15%
D = 10000	73.79%	93.4%	97.09%	97.56%	96.46%
D = 15000	73.56%	93.61%	97.204%	97.895%	97.152%



v. CONCLUSION

We have described the use of Random Indexing to language identification. Random Indexing has been used in the study of semantic vectors since 2000 (Kanerva et al. (2000); Sahlgren (2005)), and for encoding problems in graph theory (Levy & Gayler (2009)), but only now for identifying source materials. It is based on simple operations on high-dimensional random vectors: on Random Labels with 0-mean components that allow weak signals to rise above noise as the data accumulate. The algorithm works in a single pass, in linear time, with limited memory, and thus is inherently scalable, and it produces vectors that are amenable to further analysis. The experiments reported in this paper were easy for a laptop computer.

ACKNOWLEDGMENTS

We would like to thank Mr. Abbas Rahimi (Professor at SFSU) for his insightful discussion and guidance on the project. We also like to thank members on stack exchange and couple other classmates for helping us with ideas while coding.

References:

- Joshi, A. and Halseth, J.T. Github: Random indexing for languages python implementation, 2014. URL https://github.com/halseth/vs265_project_f14.
- Kanerva, P. Computing with 10,000-bit words. Proc. 52nd Annual Allerton Conference on Communication, Control, and Computing, 2014.
- Input source files and Matlab implementation file from Dr. Abbas Rahimi.
- Sahlgren, M. An introduction to random indexing. Methods and Applications of Semantic Indexing Workshop at the 7th international conference on Terminology and Knowledge Engineering , 2005.
- http://arxiv.org/abs/1412.7026