Big Data Content Analytics

Group Project

[May 2017 – Winter Semester (PT)]

Team Members
Alexandra Rousi – BAPT1536
Konstantinos Pantazis – BAPT1413
Maria Karampela – BAPT1519
Maria Choupa – BAPT1517
Myrto Perati – BAPT1501

Contents

INTRODUCTION

Text analytics is the process of converting unstructured text data into meaningful data for analysis, to measure customer opinions, product reviews, feedback, to provide search facility, sentiment analysis and entity modeling to support fact based decision making. Text analysis uses many linguistic, statistical, and machine learning techniques. Text Analytics involves information retrieval from unstructured data and the process of structuring the input text to derive patterns and trends and evaluating and interpreting the output data. It also involves lexical analysis, categorization, clustering, pattern recognition, tagging, annotation, information extraction, link and association analysis, visualization, and predictive analytics. Text analytics determine keywords, topics, category, semantics, tags from the millions of text data available in an organization in different files and formats. The term Text Analytics is roughly synonymous with text mining. (1)

During this project we will apply Text Analytics & Neural Network techniques using TensorFlow Python library. TensorFlow is an open source software library for machine learning across a range of tasks, developed by Google to meet company's needs for systems capable of building and training neural networks to detect and decipher patterns and correlations, analogous to the learning and reasoning which humans use. The data set that will be our subject of study is a fraction of crawled news' articles from online newspapers and especially from: "Eleftherotypia", "Rizospastis" and "Kathimerini", written by 22 different authors. Our scope is to:

- learn user embeddings from available past articles, and
- use these embeddings to classify new texts to specific categories, e.g. authors or article's field of interest.

The total size of text files (article corpus + metadata) is ~250MB. In more detail, the dataset contains the following characteristics for each record per different newspaper:

Eleftherotypia: category, date, description, title, author, sources, link, keywords, aid, images, and article.

Rizospastis: aid, article, author, category, date, subcategory, link, nof_words, page, title, title2, and version.

Kathimerini: aid, article, author, category, date, subcategory, and title.

article	author	category	date	description	images	keywords	link	sources	title
Ο Μάσιμο Ταρτάλια, που χτύπησε και τραυμάτισε ...	[Της ΚΑΤΕΡΙΝΑΣ ΤΖΑΒΑΡΑ]	[Διεθνή]	2009-12-16T00:00:00	Ενωτικό είναι το μήνυμα που έστειλε ο Ιταλός π...	[http://s.enet.gr/resources/2009-12/14-12-thum...	[Ιταλία]	http://www.enet.gr/?i=news.el.article&id=112816	[(Πηγές: Ασ. Πρεσς, Γαλλικό, Reuters, ΑΠΕ, Rep...	Μήνυμα αγάπης από καβαλιέρε
Η επέμβαση διήρκεσε περίπου πέντε ώρες και αφα...	[Της ΚΑΤΕΡΙΝΑΣ ΤΖΑΒΑΡΑ]	[Διεθνή]	2009-12-22T00:00:00	Ικανοποιητικά αναρρώνει από την πρώτη ευαίσθητ...	[http://s.enet.gr/resources/2009-12/12-19-thum...	[Βραζιλία]	http://www.enet.gr/?i=news.el.article&id=114363	[(Πηγές: Ασ. Πρες, Reuters)]	Φρίκη πάνω σε παιδικό σώμα!

Figure 1- Example of record

Tasks allocation

Data cleanup/evaluation/splitting: Myrto Perati & Maria Choupa
Setup working environment/TensorFlow installation: Konstantinos Pantazis
Python Implementation: Konstantinos Pantazis & Maria Karampela
Neural network modeling /techniques to be used: Alexandra Rousi
Review of Literature/relative papers: Myrto Perati & Maria Choupa
Reporting: Alexandra Rousi & Maria Karampela

Data cleanup/evaluation/splitting

Obtain raw data

The raw dataset has been collected by Dimitris Pappas during with research work. We thank with for providing the processed data in a ready-to-use format (pickles)

Data Cleanup

We have proceeded with preprocessing our data set to clean, evaluate and prepare our data input. We have dropped the additional characteristics that wouldn't add value to our model and project scope, for instance we have dropped 'aid','date','link','nof_words','page','version','title2' from rizospastis data set and we have renamed the characteristics ('subcategory' to 'category') so as to have a common data structure. Eventually and after evaluating and structuring our model, we have decided to drop also 'category' and 'title', since we will work with the 'article' and the 'author'. These will be the data to train our model and create user embeddings.

	article	author	category	title
0	Τώρα αρχίζει η μεγάλη μάχη! Ή μήπως αισιοδοξο...	kanelli	politics	Εργασία μάρκα Vs Κεφάλαιο μπάνκα
1	Οταν ο λαός γρυλίζει και μάλιστα μαζικά κι εκκ...	kanelli	politics	Ανάλωσέ το...
2	Το πιο φτηνό, το πιο επικίνδυνα άχρηστο παραμύ...	kanelli	politics	Η φόλα
3	Η παγκοσμιοποίηση δεν είναι βεβαίως μια καινού...	kanelli	politics	«Π» όπως πόρνη και πλιάτσικο
4	Επί χούντας, εκεί που η τηλεόραση ήταν στα σπά...	kanelli	politics	Η σουπιά

Figure 2-Data Structure

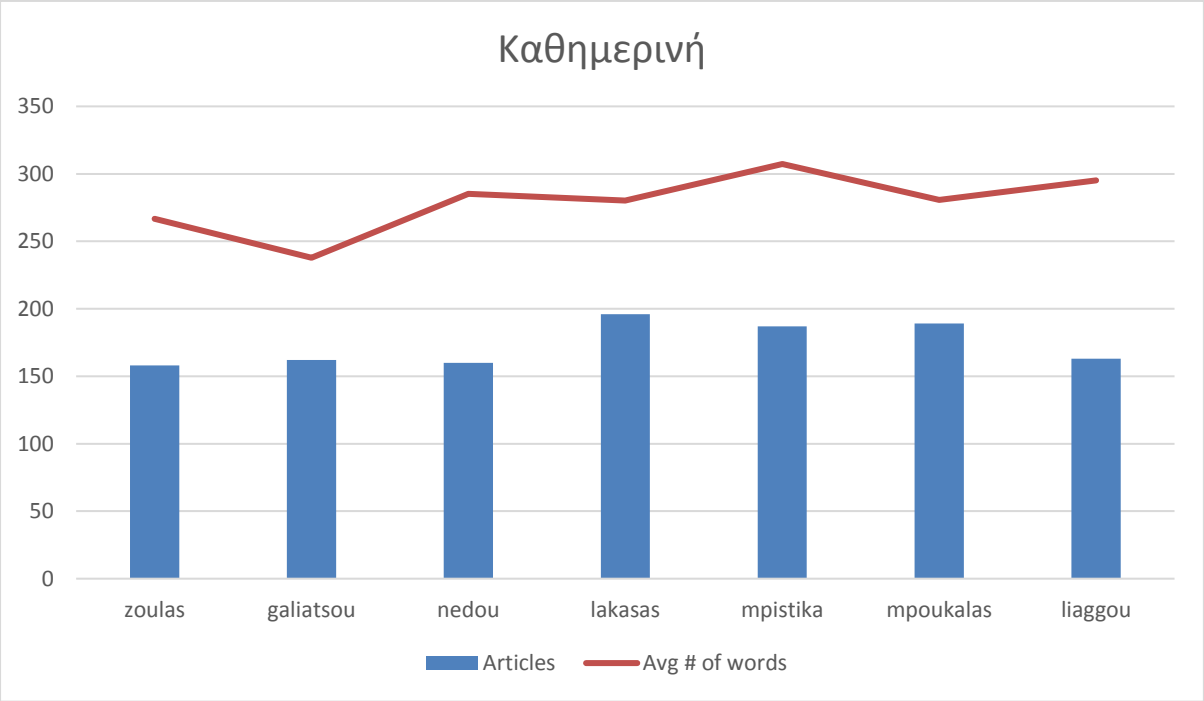
	article	author
0	Το 1844 ο Κωνσταντίνος Ιωνίδης προσέφερε στον ...	lakasas
1	«Η διεύρυνση των εργασιακών σχέσεων συναντάται...	lakasas
2	Τη μείωση του αριθμού των εισακτέων σε σχολές ...	lakasas
3	«Η επαναλειτουργία της Θεολογικής Σχολής της Χ...	lakasas
4	Κατά δύο αυξάνονται από την επόμενη σχολική χρ...	lakasas

Figure 3- Final data structure

Performing some simple statistics, we count the number of articles per author and the average number of words per article. Here there are our results:

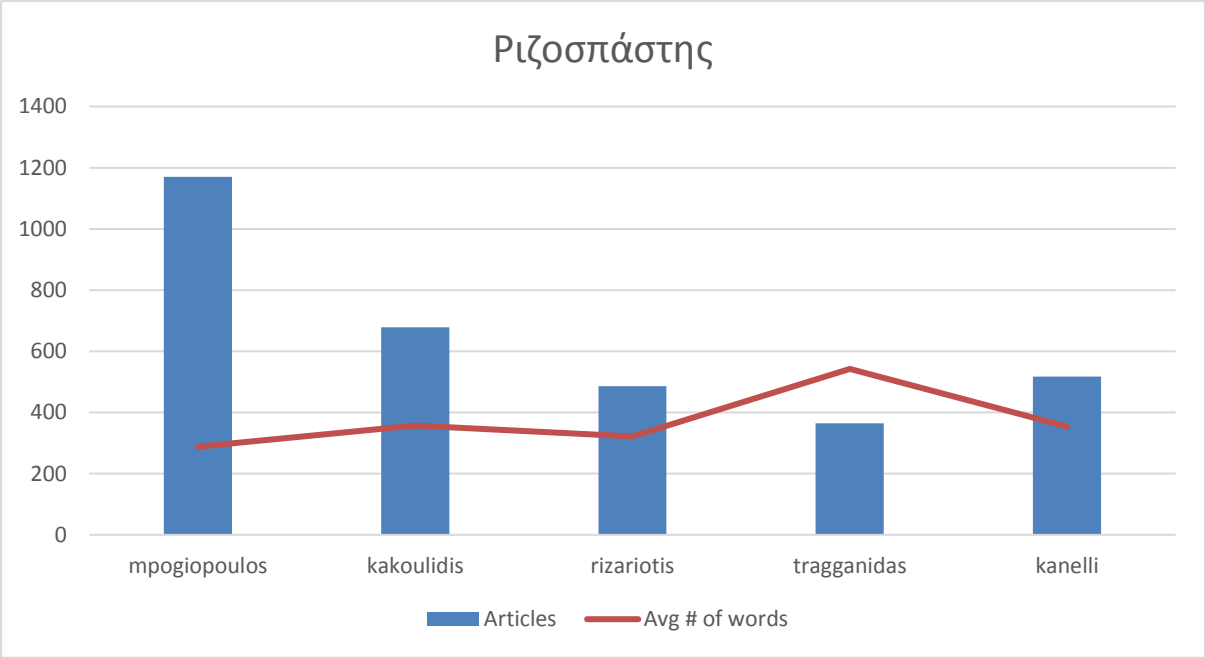
```
Author: Του Κωνσταντινου Ζουλα - 158 articles - avg words: 266.626582278481
Author: Του Παναγη Γαλιατσατου - 162 articles - avg words: 237.820987654321
Author: Του Βασιλη Νεδου - 160 articles - avg words: 285.225
Author: Του Αποστολου Λακασα - 196 articles - avg words: 280.30102040816325
Author: Της Ελενης Μπιστικα - 187 articles - avg words: 307.28877005347596
Author: Του Παντελη Μπουκαλα - 189 articles - avg words: 280.7037037037037
Author: Της Χρυσας Λιαγγου - 163 articles - avg words: 295.04294478527606
```

Figure 4--No of Kathimerini articles



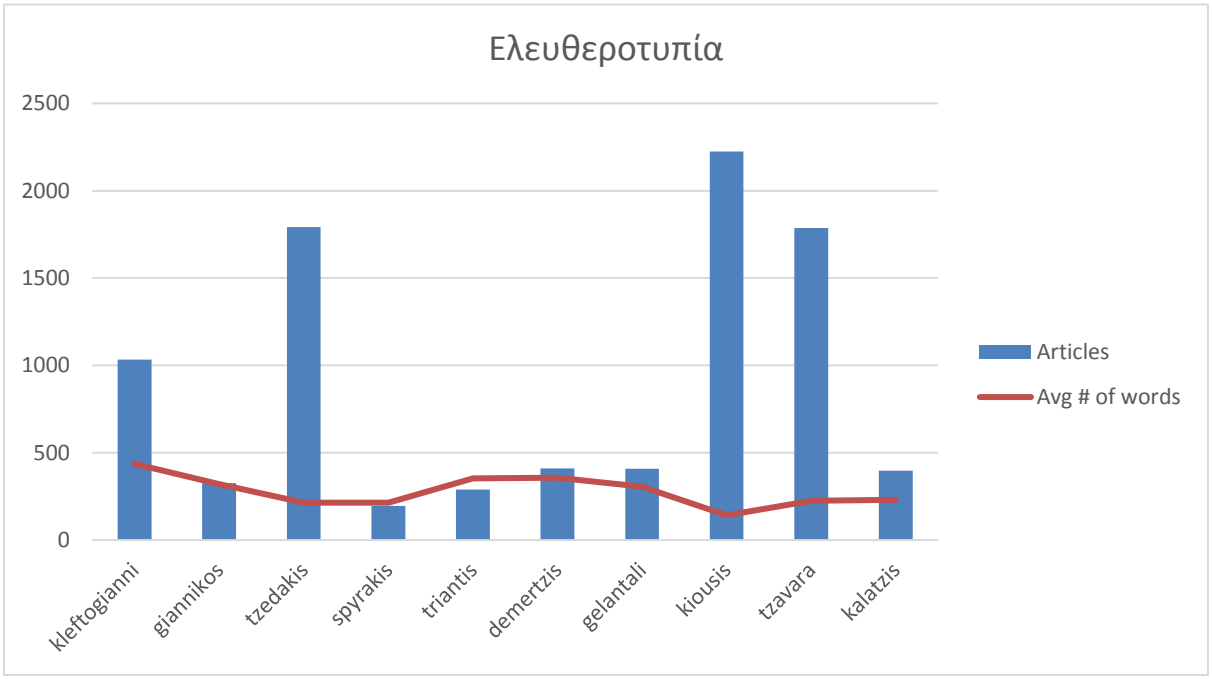

```
Author: Νίκος ΜΠΟΓΙΟΠΟΥΛΟΣ - 1170 articles - avg words: 288.24957264957266
Author: Του Γιώργου ΚΑΚΟΥΛΙΔΗ - 678 articles - avg words: 357.26106194690266
Author: Παύλος ΡΙΖΑΡΓΙΩΤΗΣ - 486 articles - avg words: 321.2860082304527
Author: Γρηγόρης ΤΡΑΓΓΑΝΙΔΑΣ - 364 articles - avg words: 542.3489010989011
Author: Λιάνας ΚΑΝΕΛΛΗ - 517 articles - avg words: 354.52611218568666
```

Figure5- No of Rizospastis articles




```
Author: ['Της ΙΩΑΝΝΑΣ ΚΛΕΦΤΟΓΙΑΝΝΗ'] - 1034 articles - avg words: 436.1866537717602
Author: ['Του ΑΔΑΜ ΓΙΑΝΝΙΚΟΥ'] - 326 articles - avg words: 320.35276073619633
Author: ['Υπεύθυνος: ΕΠΙΜΕΛΕΙΑ: ΓΙΩΡΓΟΣ ΤΖΕΔΑΚΙΣ / tzedakis@enet.gr'] - 1793 articles - avg words: 214.14110429447854
Author: ['Υπεύθυνος: Βαγγέλης Σπυράκης'] - 195 articles - avg words: 213.23076923076923
Author: ['Του ΓΙΑΝΝΗ ΤΡΙΑΝΤΗ'] - 289 articles - avg words: 352.8200692041523
Author: ['Του ΑΡΓΥΡΗ ΔΕΜΕΡΤΖΗ'] - 410 articles - avg words: 357.280487804878
Author: ['Του ΜΙΧΑΗΛ ΓΕΛΑΝΤΑΛΙ'] - 409 articles - avg words: 305.28606356968214
Author: ['Υπεύθυνος: ΕΠΙΜΕΛΕΙΑ: ΓΙΩΡΓΟΣ ΚΙΟΥΣΗΣ kiousis@enet.gr\nΑπό τον ΠΕΤΡΟ ΜΑΝΤΑΙΟ'] - 2225 articles - avg words: 141.8840
494382021
Author: ['Της ΚΑΤΕΡΙΝΑΣ ΤΖΑΒΑΡΑ'] - 1787 articles - avg words: 224.7901510912143
Author: ['Του ΜΑΝΩΛΗ ΚΑΛΑΤΖΗ'] - 398 articles - avg words: 230.82663316582915
```

Figure 6-No of Eleftherotypia articles



At a second stage of our data preprocessing procedure, is to find the most commonly used words per author and to decide whether we will keep them so as to train our model or not. An example of the most commonly used words of Mpogiopoulos, is displayed below:

```
Data size 680107
Most common words (+UNK) [['UNK', 10187], (',', 42329), ('.', 20198), ('και', 18057), ('του', 15599), ('το', 13897), ('της', 12
810), ('να', 11596), ('που', 11048), ('την', 8741), ('των', 7659), ('για', 7363), ('η', 7356), ('από', 7247), ('τους', 7048),
 ('με', 6769), ('τα', 6753), ('ο', 6746), ('είναι', 6261), (')', 5585)]
Sample data [54] ['ΜΠΟΓΙΟΠΟΥΛΟΣ']
```

Figure 5-Common words example

Evaluation

During the first parsing of our dataset, we quickly observed, as anticipated, that the most used words of all users where common words such as και, η, το, που, and also symbols such as . , ! ( ) ; / * In our approach, we will consider the occurrence of these common words as stopwords and we will use a dictionary of greek stopwords[1] to remove them from corpus.

To use the stopwords list effectively, we had to transform our dataset to lower case and remove stress from the words.

Splitting part

We used all available data to learn user embeddings for this project. For prediction tasks we should have split the available data to training/testing sets but this has been intentionally left out of scope for this project.

Setup working environment/TensorFlow installation - Python Implementation

Our project implementation was done in Python (version 3.6), using Jupyter notebooks. We needed to have access to our notebooks from the Internet, so we created some VMs in Okeanos cloud service. Each team member was assigned a Linux server (Ubuntu 16.04.2 LTS) with 8GB RAM and 4 CPU cores (no GPU ☹). Tensorflow version is 1.0.1

Our first model is based on word2vec implementation [2]

[1] https://github.com/xtsimpouris/gr-nlp-law/blob/master/Greek%20Stopwords/stopwords.txt
[2] https://github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/examples/tutorials/word2vec/word2vec_basic.py

Let's call this **author2vec** :) The main focus here was to learn author(user) embeddings by trying to predict the author that wrote a specific word, for all words in our corpus. By the end of the training process, the tensor of user embeddings will be available as a by-product of this trivial task.

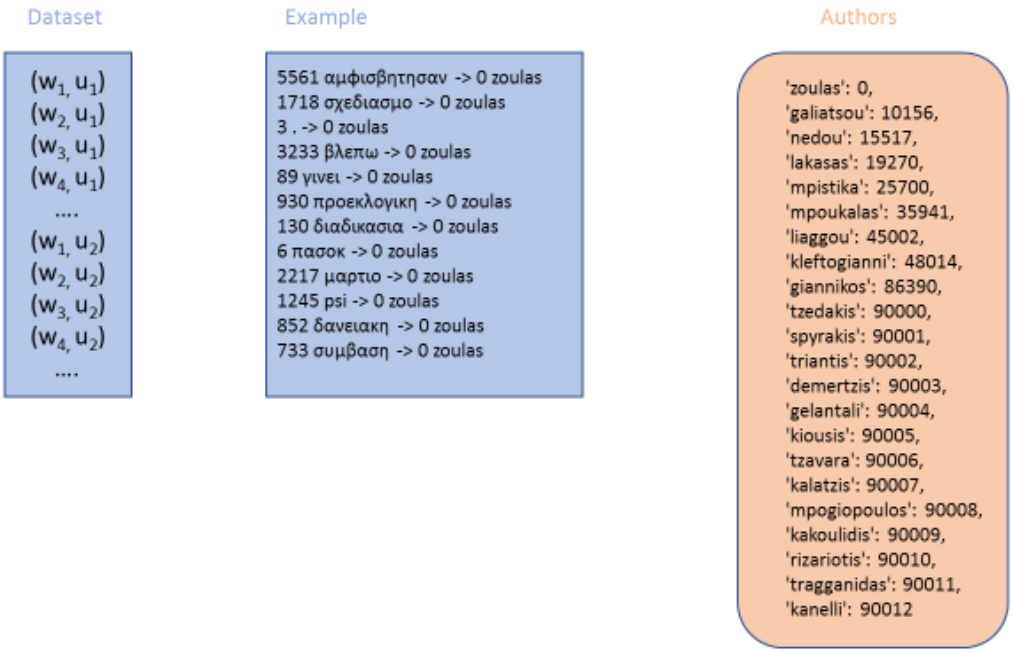
We took the simplest form of skip-gram model by using just one input word with one target.

Author2vec steps:

Step 1: Read data (articles, author) for all newspapers from pickled files on the disk.

Step 2: Tokenize article text to words using word_tokenize module from nltk package

Step 3: Build the dictionary: we consider a dictionary with size 90000 words. (At this step we transform to lower case and remove stopwords from corpus). For each author, we keep the N most used words and we add the new words to our dictionary. Finally, we create tuples of (author_index, word_index) as our final dataset. Author names are considered as words and have an index within our word dictionary. To have access to the author indices, we also store their values to a separate authors dictionary.



Step 4: Function to generate a training batch for our model. Essentially, it keeps a global index of the current batch and returns the (predefined number of) next batch_size tuples. Once all tuples in the dataset have been consumed by our model, we stat feeding again from the start of our dataset. This point defines a new epoch.

Step 5: Build and train a skip-gram model. We would have a dimension of the embedding vector. Also, how many times to reuse an input to generate a label and how we pick a random validation set to sample nearest neighbors. Here we limit the validation samples to the words that have a low numeric ID, which by construction are also the most frequent. Look up embeddings for inputs, construct the variables for the NCE loss, compute the average NCE loss for the batch and finally compute the cosine similarity between minibatch examples and all embeddings.

Step 6: Begin training. With a num_steps = 74001 (4 epochs having embedding size 200) we perform one update step by evaluating the optimizer op.

Step 7: Finally we visualize the produced embeddings using tSNE.

* We save the learned embeddings in the disk for future use.

Neural network modeling /techniques to be used

In Deep Learning, Recurrent Neural Networks (RNN) are a family of neural networks that excels in learning from sequential data. A class of RNN that has found practical applications is Long Short-Term Memory (LSTM) because it is robust against the problems of long-term dependency.

Vector Representations of Words

Word2vec is an algorithm for constructing vector representations of words, also known as *word embeddings*. The vector for each word is a semantic description of how that word is used in context, so two words that are used similarly in text will get similar vector representations. Once you map words into vector space, you can then use vector math to find words that have similar semantics.

Recurrent neural network

A **recurrent neural network** (**RNN**) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition or speech recognition.
The most straight-forward example is perhaps a time-series of numbers, where the task is to predict the next value given previous values. The input to the RNN at every time-step is the *current value* as well as a *state vector* which represent what the network has "seen" at time-steps before. This state-vector is the encoded memory of the RNN, initially set to zero.

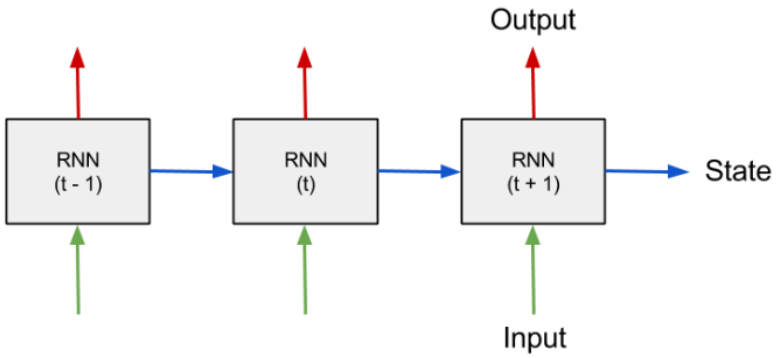


Figure 6:Schematic of a RNN processing sequential data over time.

The goal of the problem is to fit a probabilistic model which assigns probabilities to sentences. It does so by predicting next words in a text given a history of previous words.

Recurrent neural networks are networks with loops in them, allowing information to persist. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

LSTM Networks

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.[1] They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

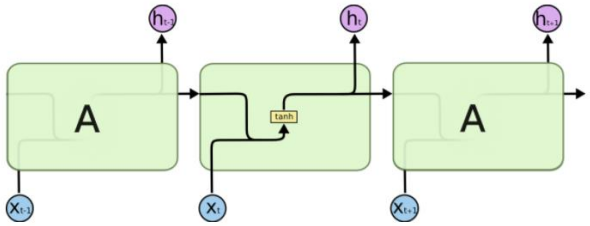


Figure 7:The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

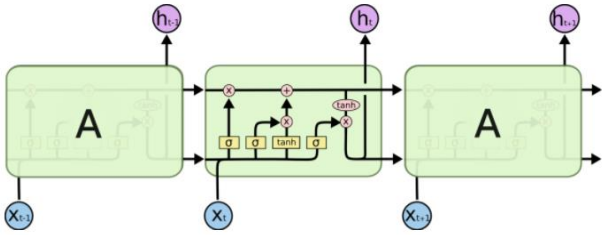


Figure 8:The repeating module in an LSTM contains four interacting layers.

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"

An LSTM has three of these gates, to protect and control the cell state.

The core of the model consists of an LSTM cell that processes one word at a time and computes probabilities of the possible values for the next word in the sentence. The memory state of the network is initialized with a vector of zeros and gets updated after reading each word.

General view about other Neural Networks and Algorithms:

Keras:

Keras is an open source neural network library written in Python. Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle itself low-level operations such as tensor products, convolutions and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras.

At this time, Keras has two backend implementations available: the **TensorFlow** backend and the **Theano** backend.

- For this project we use **Tensorflow** which is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

Theano:

- **Theano** is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs. Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations. This combination of CAS with optimizing compilation is particularly useful for tasks in which complicated mathematical expressions are evaluated repeatedly and evaluation speed is critical. For situations where many different expressions are each evaluated once Theano can minimize the amount of compilation/analysis overhead, but still provide symbolic features such as automatic differentiation.

Also for the **Word embedding** there are some softwares for training and using word embeddings. For example Tomas Mikolov's Word2vec, Stanford University's GloVe, Gensim and Deeplearning4j. Principal Component Analysis (PCA) and T-Distributed Stochastic Neighbour Embedding (t-SNE) are both used to reduce the dimensionality of word vector spaces and visualize word embeddings and clusters.

For this project we use Word2Vec but let's see some things about GloVe and Gensim.

GloVe:

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. It is developed as an open-source project at Stanford.

But how is GloVe different from word2vec?

Both models learn geometrical encodings (vectors) of words from their co-occurrence information (how frequently they appear together in large text corpora). **They differ in that word2vec is a "predictive" model, whereas GloVe is a "count-based" model.**

Predictive models learn their vectors in order to improve their predictive ability of Loss(target word | context words; Vectors).

Count-based models learn their vectors by essentially doing dimensionality reduction on the co-occurrence counts matrix. They first construct a large matrix of (words x context) co-occurrence information, i.e. for each "word" (the rows), you count how frequently we see this word in some "context" (the columns) in a large corpus. The number of "contexts" is of course large, since it is essentially combinatorial in size. So then they factorize this matrix to yield a lower-dimensional (word x features) matrix, where each row now yields a vector representation for each word. In general, this is done by minimizing a "reconstruction loss" which tries to find the lower-dimensional representations which can explain most of the variance in the high-dimensional data. In the specific case of GloVe, the counts matrix is preprocessed by normalizing the counts and log-smoothing them. This turns out to be A Good Thing in terms of the quality of the learned representations.

However, as pointed out, when we control for all the training hyper-parameters, the embeddings generated using the two methods tend to perform very similarly in downstream NLP tasks. The additional benefits of GloVe over word2vec is that it is easier to parallelize the implementation which means it's easier to train over more data, which, with these models, is always A Good Thing.

Gensim:

Gensim is a mature open-source vector space modeling and topic modeling toolkit implemented in Python. It uses NumPy, SciPy and optionally Cython for performance. It is specifically designed to handle large text collections, using data streaming and efficient incremental algorithms, which differentiates it from most other scientific software packages that only target batch and in-memory processing.

But how is Gensim different from word2vec?

Gensim is not a technique itself. Gensim is a NLP package that contains efficient implementations of many well-known functionalities for the tasks of topic modeling, on the other hand Word2Vec is an implementation of the Skip gram model for building "word" vector representations i.e. taking a huge corpus and producing vector of real numbers for each unique "word" which afterwards can be used for many NLP applications. Also, Gensim even has ported word2vec inside it with additional functionality.

Review of Literature

Distributed Representations of Sentences and Documents

Text classification and clustering techniques play an important role in many applications, e.g. document retrieval. At the heart of these applications is machine learning algorithms such as logistic regression or K-means. Bag-of-words (BOW) is the most common fixed-length feature vector which suffers from work order loss, data scarcity, high dimensionality and semantics ignorance. Here, we will try to implement Paragraph Vector algorithm as this has been introduced by Quoc Le and Tomas Mikolov of Google Inc. Paragraph Vector is an unsupervised framework that learns continuous distributed vector representations for pieces of texts. The texts can be of variable length, ranging from sentences to documents. Here, we will concatenate the paragraph vector with several word vectors from a paragraph. While paragraph vectors are unique among paragraphs, the word vectors are shared.

Learning Vector Representation of Words

In this framework, every word is mapped to a unique vector, represented by a column in a matrix W. The column is indexed by position of the word in the vocabulary. The concatenation or sum of the vectors is then used as features for prediction of the next word in a sentence. More formally, given a sequence of training words: w1, w2, w3, …, the objective of the word vector model is to maximize the average log probability that next word will be w_T given a sequence of other words. The neural network based word vectors are usually trained using stochastic gradient descent where the gradient is obtained via back propagation. After being trained, the paragraph vectors can be used as features for the paragraph. They feed these features directly to conventional machine learning techniques such as logistic regression, support vector machines or K-means. After the training converges, words with similar meaning are mapped to a similar position in the vector space.

Siamese CBOW: OptimizingWord Embeddings for Sentence Representations

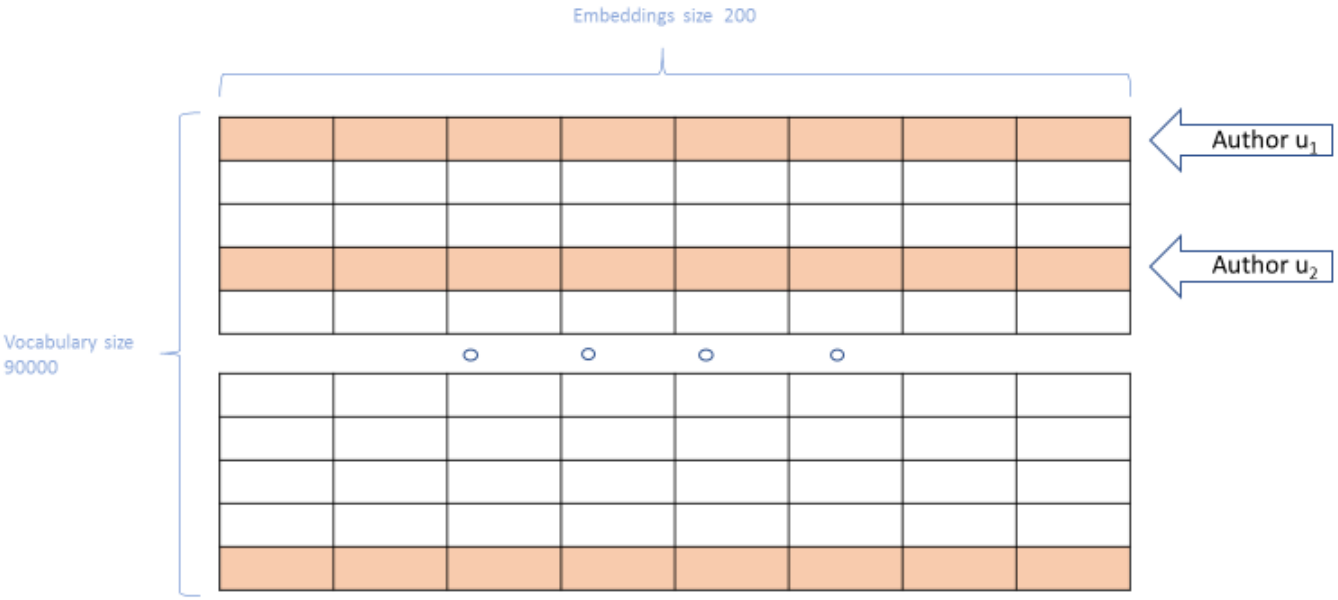
In the paper of Kenter, Borisov & Rijke (2016) an alternative to the “word2vec” methodology is presented. The word2vec is an algorithm where a supervised training criterion for obtaining word embeddings from unsupervised data, by predicting, for every word, its surrounding words, is constructed. In this paper, the same strategy is applied to the sentence level where the goal is to predict a sentence from its adjacent sentences. This methodology could be a second stage to the project that is presented here. In a few words, the Siamese CBOW trains word embeddings directly for the purpose of being averaged and the underlying neural network learns word embeddings by predicting, from a sentence representation, its surrounding sentences. The authors do this by directly including a comparison of sentence embeddings—the averaged embeddings of the words they contain—in the cost function of a network. Moreover, they aim to optimize word embeddings for sentence representations as it is typically done for word embeddings; by making it possible to train their embeddings fast and in a scalable way from unlabeled training data as well as to train them in a way that will be able to be employed across different tasks.

In a few words, the methodology they use to train their embeddings is presented. The goal of their training process is to construct a supervised training criterion by having their network predict sentences that occur next to each other in the training data. The input in their process is a projection layer that selects embeddings from a word embedding matrixW (that is shared across inputs) for a given input sentence. The word embeddings are averaged in the next layer, which yields a sentence representation with the same dimensionality as the input word embeddings. The cosine similarities between the sentence representation for each sentence and the other sentences are calculated in the penultimate layer and a softmax is applied in the last layer to produce the final probability distribution. In the end, only the weights in the word embedding matrix are the only trainable parameters in the Siamese CBOW network.

In the end of their paper, the authors conclude that Siamese CBOW is a fast and reliable algorithm for generating word embeddings optimized for being averaged across sentences.

Reporting – Conclusions

We created a word vocabulary of 90.000 words. Embedding vector size set to 200. As noted before, we used all available articles of each author. After processing, our dataset contained 3.697.025 tuples (word, author)



The following output shows the training process for 74.000 steps / 4 epochs:

--

Initialized

Average loss at step 0 : **306.557922363**

Average loss at step 2000 : 151.290333273

Average loss at step 4000 : 84.1221010342

Average loss at step 6000 : 52.5247837538

Average loss at step 8000 : 37.1424338791

Average loss at step 10000 : 27.7675384989

Average loss at step 12000 : 21.4129059091

Average loss at step 14000 : 16.0890726156

Average loss at step 16000 : 13.3901821244

Average loss at step 18000 : 10.1815554794

Epoch 1 just finished!

Average loss at step 20000 : 11.8884472606

Average loss at step 22000 : 9.9731038982

Average loss at step 24000 : 6.97735548498

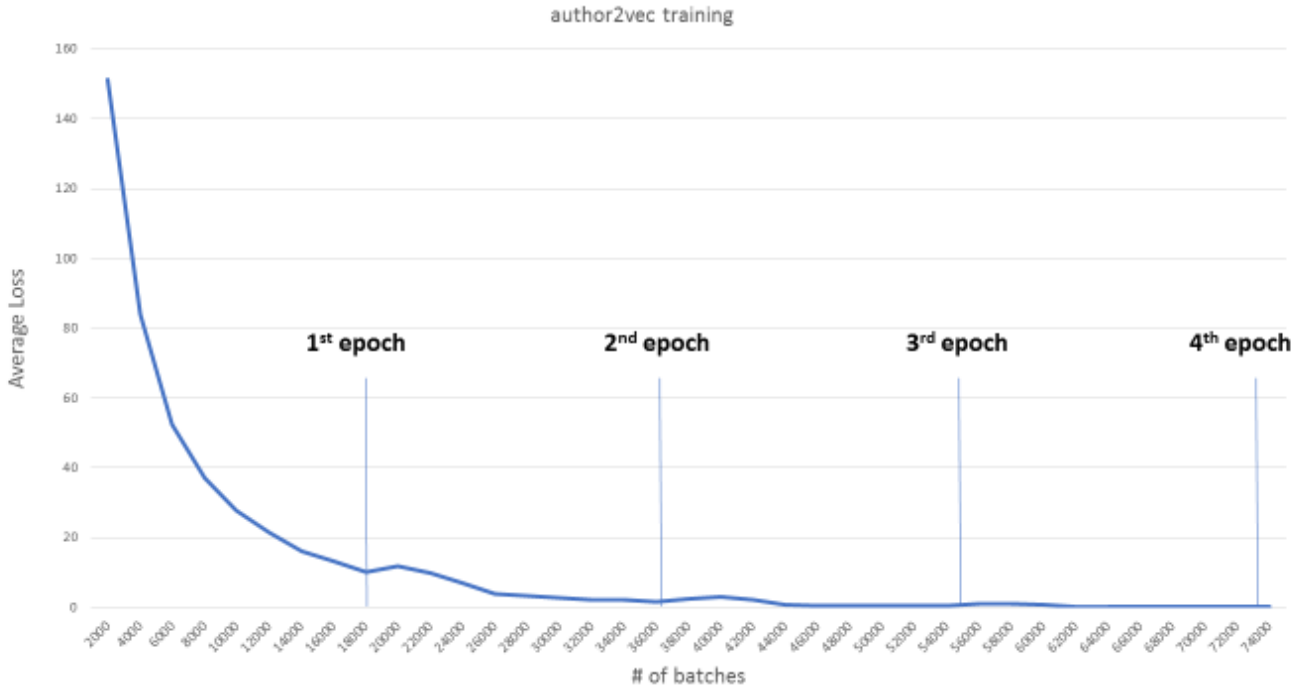
Average loss at step 26000 : 3.89015097793

Average loss at step 28000 : 3.36753835666

Average loss at step 30000 : 2.71839862359

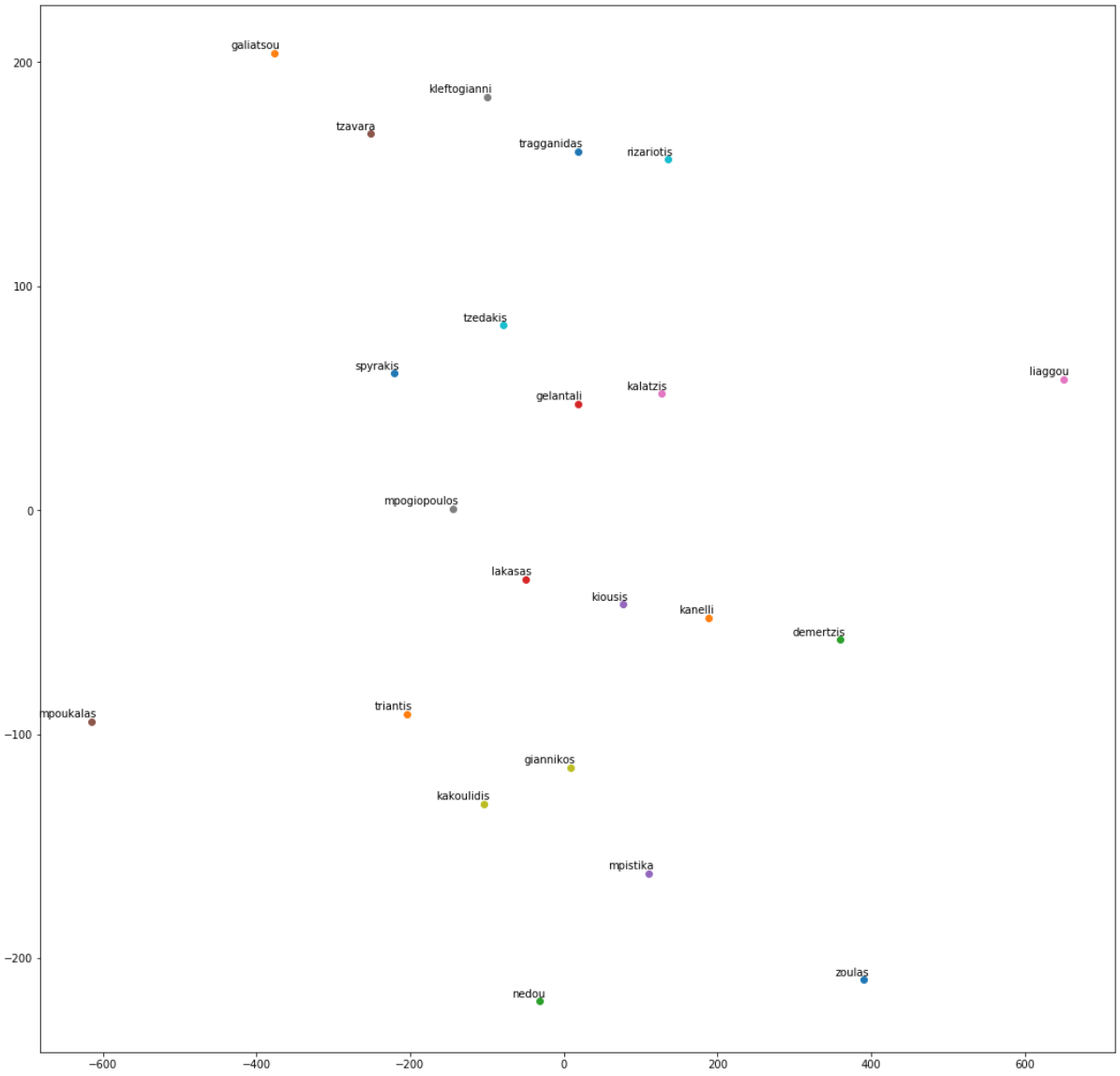
Average loss at step 32000 : 2.28612280754

Average loss at step 34000 : 2.1091589355
Average loss at step 36000 : 1.60069775732
Epoch 2 just finished!
Average loss at step 38000 : 2.54768097164
Average loss at step 40000 : 2.97829922005
Average loss at step 42000 : 2.12709161307
Average loss at step 44000 : 0.877558657598
Average loss at step 46000 : 0.623970782024
Average loss at step 48000 : 0.561257907672
Average loss at step 50000 : 0.495338672839
Average loss at step 52000 : 0.420467982602
Average loss at step 54000 : 0.417875091807
Epoch 3 just finished!
Average loss at step 56000 : 0.993680757665
Average loss at step 58000 : 1.00570583211
Average loss at step 60000 : 0.879357251156
Average loss at step 62000 : 0.188788442416
Average loss at step 64000 : 0.209896088698
Average loss at step 66000 : 0.149019773626
Average loss at step 68000 : 0.125519577501
Average loss at step 70000 : 0.0849295024187
Average loss at step 72000 : **0.104189667971**
Epoch 4 just finished!
Average loss at step 74000 : 0.296238166741



The following visual representation of learned embeddings should be studied further to show whether it succeeds in capturing the authoring style of our authors under test.

Note: we did not try to explain this output based on the political background of the newspaper that host author's articles. This would be more like a topic discovery problem, rather than author style recognition.



Future work

Of course, the presented outputs are not enough to account for a stable system that can be used for accurate predictions and inferences. Some proposals for further study would be:

- More NN techniques (RNN/LSTM, …) and libraries/frameworks should be tested (keras, gensim, glove)
- Word stemming would help in dramatical reduction of vocabulary size, allowing more words to be included in training.
- n-grams can be considered instead of single word samples
- After having successfully learned author embeddings:
    - Predict the next words of a new text, having identified the authoring style
    - Identify topics of interest for each author

Bibliography

1. PREDICTIVE ANALYTICS TODAY. [Ηλεκτρονικό] http://www.predictiveanalyticstoday.com/text-analytics/.

2. [Ηλεκτρονικό]

3.Exploring Word Embeddings and Character N-Grams for Author Clustering http://ceur-ws.org/Vol-1609/16090984.pdf

4. Author2Vec: Learning Author Representations by Combining Content and Link Information https://researchweb.iiit.ac.in/~soumyajit.ganguly/papers/A2v_1.pdf

5. Author Profiling using SVMs and Word Embedding Averages http://ceur-ws.org/Vol-1609/16090815.pdf

6. A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification https://arxiv.org/pdf/1510.03820.pdf

7. http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

8. http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/

9. A Persona-Based Neural Conversation Model https://nlp.stanford.edu/pubs/jiwei2016Persona.pdf

10. Distributed Representations of Sentences and Documents https://cs.stanford.edu/~quocle/paragraph_vector.pdf

11. Siamese CBOW: Optimizing Word Embeddings for Sentence Representations http://www.aclweb.org/anthology/P16-1089

12. Word2vec: https://www.tensorflow.org/tutorials/word2vec

13. RNN model https://medium.com/@erikhallstrm/hello-world-rnn-83cd7105b767

14. https://www.tensorflow.org/tutorials/recurrent

15. LSTM model describe: http://colah.github.io/posts/2015-08-Understanding-LSTMs/