

Lecture 08

Introduction to NLP and Word2Vec

CSCI E-89 Deep Learning, Fall 2024

Zoran B. Djordjević

References

This lecture follows material presented in:

- **TensorFlow example:** <https://www.tensorflow.org/text/tutorials/word2vec>
- **Deep Learning, NLP, and Representations** 2014, by Christopher Olah
<http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/>
- Parts of Chapter 13, Hands on Machine Learning with Scikit-learn & TensorFlow, by Aurélien Géron, O'Reilly, 2nd and 3rd Edition, 2019 and 2023
- Parts of Chapter 6, Deep Learning in Python by Francois Chollet, O'Reilly 2017
- Parts of Chapter 11, Deep Learning in Python, 2nd Ed. by Francois Chollet, O'Reilly 2021

Perhaps the most important general reference on NLP, Speech and Language Processing is:

**An Introduction to Natural Language Processing,
Computational Linguistics, and Speech Recognition
Second Edition**

by [Daniel Jurafsky](#) and [James H. Martin](#)

<http://www.cs.colorado.edu/~martin/slp2.html>

- The third edition of this book is in preparation. You can download most of its content free online
- <https://web.stanford.edu/~jurafsky/slp3/>

Uses of NLP

- Natural Language Processing is the field of Computer Science behind:
 - Google and any other text and speech search
 - Document Retrieval
 - Text classification or categorization
 - Name Entity Recognition, i.e., places, companies and persons identification
 - Machine language translation
 - Sentiment analysis
 - Speech processing and understanding
 - Automated customer service
 - Automated news generation
 - Question-answer systems
 - ChatGPT
 - LLMs
 -

"Classical" NLP APIs

NLP Frameworks

- Deep Learning (DL) for Natural Language Processing (NLP) uses neural networks to analyze and classifies words, sentences and paragraphs.
- Besides Deep Learning tools and APIs there are several mostly Python NLP frameworks.
- An impression is that **NLTK**, and **spaCy** are the most popular.
 - NLTK is somewhat last century but has many useful features and is still used.
 - spaCy is more modern but with smaller feature set.
 - Gensim has the smallest feature set. It has some rare functions in word2vec domain.
 - SciKit-Learn is a broader Machine Learning library which dropped its NLP module but is useful anyway.
 - **FastText** by Facebook is also an excellent toolkit and is getting very popular.
- You need the above Python frameworks, so you do not end-up coding everything in Keras, TensorFlow or PyTorch. Those Python frameworks are most useful for data preparation, cleaning, initial analysis.
- There are still around C, C++ and Java based NLP frameworks. They are not as popular as the Python packages.

NLTK.org, version 3.9.1

- NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to [over 50 corpora and lexical resources](#) such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active [discussion forum](#).
- Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.
- NLTK has been called “a wonderful tool for teaching, and working in, computational linguistics using Python,” and “an amazing library to play with natural language.”
- [The book Natural Language Processing with Python](#) provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more. The online version of the book has been updated for Python 3 and NLTK 3. (The original Python 2 version is still available at https://www.nltk.org/book_1ed.)

Some things you can do with NLTK

- **Tokenize and tag some text:**

```
>>> import nltk
>>> sentence = """At eight o'clock on Thursday morning
... Arthur didn't feel very good."""
>>> tokens = nltk.word_tokenize(sentence)
>>> tokens
['At', 'eight', 'o'clock', 'on', 'Thursday', 'morning',
'Arthur', 'did', 'n't', 'feel', 'very', 'good', '.']
>>> tagged = nltk.pos_tag(tokens)
>>> tagged[0:6]
[('At', 'IN'), ('eight', 'CD'), ('o'clock', 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

- **Identify named entities:**

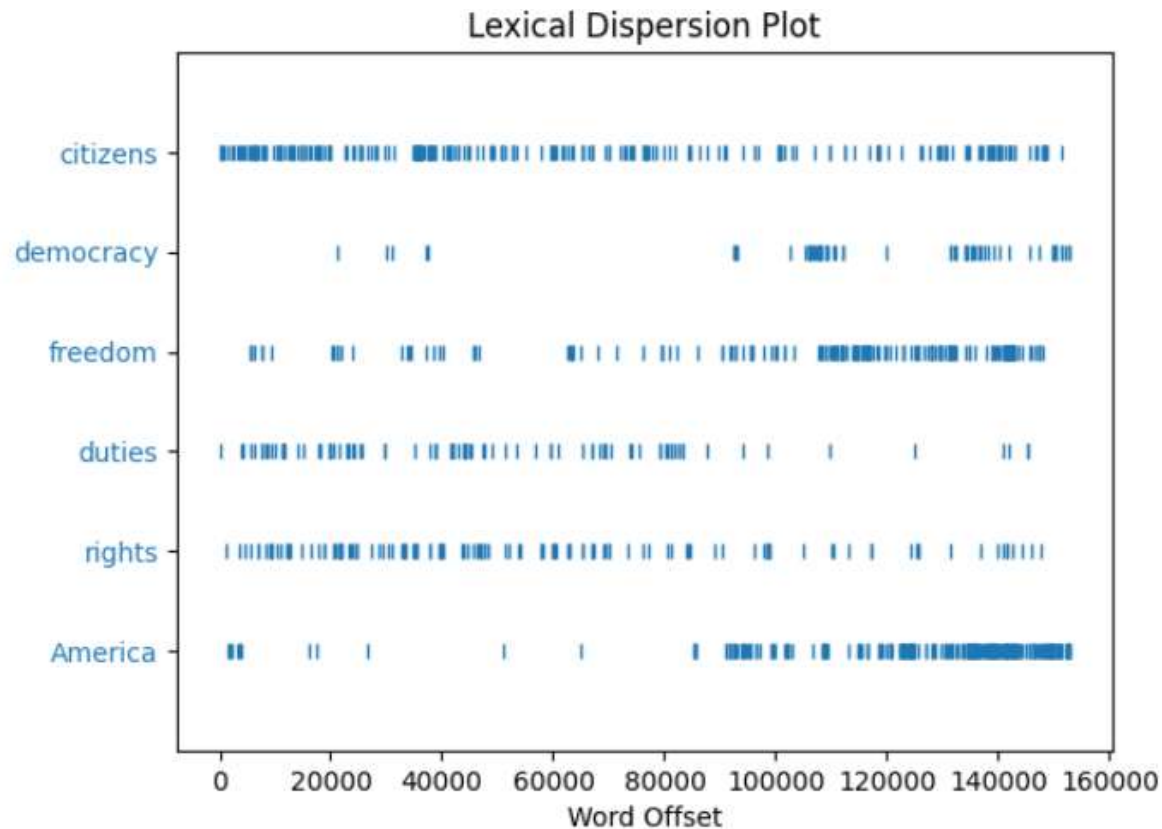
```
>>> entities = nltk.chunk.ne_chunk(tagged)
>>> entities
Tree('S', [('At', 'IN'), ('eight', 'CD'), ('o'clock', 'JJ'),
           ('on', 'IN'), ('Thursday', 'NNP'), ('morning', 'NN'),
           Tree('PERSON', [('Arthur', 'NNP')]),
           ('did', 'VBD'), ('n't', 'RB'), ('feel', 'VB'),
           ('very', 'RB'), ('good', 'JJ'), ('.', '.')]])
```

Dispersion Plot

- Determines the location of a word in text. i.e., how many words from the beginning does it appear and displays in a dispersion plot.
- Each stripe represents an instance of a word, and each row represents the entire text . This is an artificial text constructed by joining the texts of the Inaugural Address Corpus end-to-end.

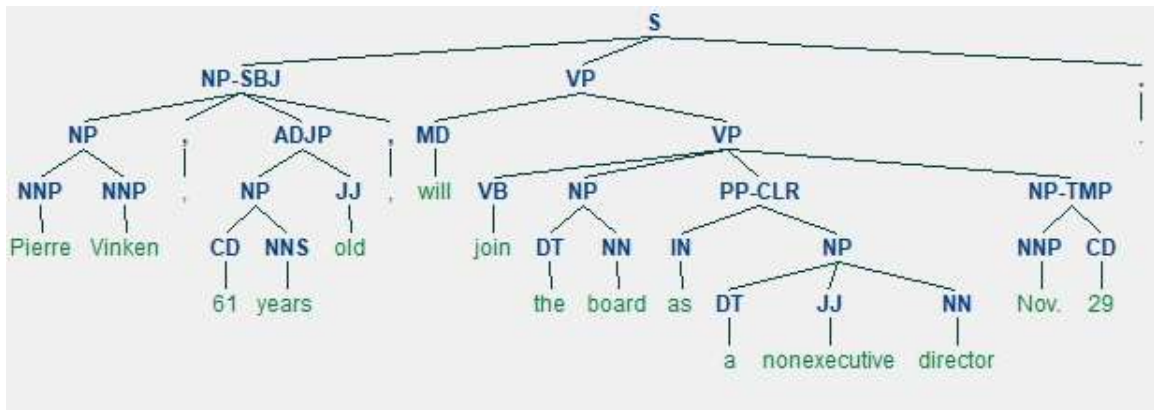
```
#text4: Inaugural Address Corpus
```

```
text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "rights", "America"])
```



Display Structure of a Sentence

```
from nltk.corpus import treebank
t = treebank.parsed_sents('wsj_0001.mrg')[0]
t.draw()
```



A note on installation of NLTK, Gensim and spaCy.

- NLTK and Gensim are moderately old packages. They do not fit well in an environment that supports spaCy. Our recommendation is that, at least initially, you create one conda virtual environment for NLTK and Gensim and a separate virtual environment for spaCy.
- In the lab notebooks, we will provide more detailed installation instructions. It appears that is better to install spaCy using conda commands like:

```
conda install -c conda-forge spacy
```

spaCy

USAGE MODELS API UNIVERSE Search docs

Industrial-Strength Natural Language Processing

IN PYTHON

Fastest in the world

spaCy excels at large-scale information extraction tasks. It's written from the ground up in carefully memory-managed Cython. Independent research has confirmed that spaCy is the fastest in the world. If your application needs to process entire web dumps, spaCy is the library you want to be using.

FACTS & FIGURES

Get things done

spaCy is designed to help you do real work — to build real products, or gather real insights. The library respects your time, and tries to avoid wasting it. It's easy to install, and its API is simple and productive. We like to think of spaCy as the Ruby on Rails of Natural Language Processing.

GET STARTED

Deep learning

spaCy is the best way to prepare text for deep learning. It interoperates seamlessly with TensorFlow, PyTorch, scikit-learn, Gensim and the rest of Python's awesome AI ecosystem. With spaCy, you can easily construct linguistically sophisticated statistical models for a variety of NLP problems.

READ MORE

Gensim

<https://radimrehurek.com/gensim/index.html>



The screenshot shows the Gensim website homepage. At the top, there's a navigation bar with links: Home, Tutorials, Install, Support, API, and About. The main header features the Gensim logo, the tagline "topic modelling for humans", and a "Download" button for the latest version from the Python Package Index. Below the header, a large blue banner states "Gensim is a FREE Python library" and lists three key features: Scalable statistical semantics, Analyze plain-text documents for semantic structure, and Retrieve semantically similar documents. To the left of this banner is a code block showing a Python script for using Gensim. Below the banner, a "Features" section highlights Scalability, Platform independent, Robust, and Open source (under the LGPL license).

Fork me on GitHub

 **gensim**
topic modelling for humans

Download
latest version from the Python Package Index

Direct install with:
easy_install -U gensim

Home Tutorials Install Support API About


```
>>> from gensim import corpora, models, similarities
>>>
>>> # Load corpus iterator from a Matrix Market file on disk.
>>> corpus = corpora.MmCorpus('/path/to/corpus.mm')
>>>
>>> # Initialize Latent Semantic Indexing with 200 dimensions.
>>> lsi = models.LsiModel(corpus, num_topics=200)
>>>
>>> # Convert another corpus to the latent space and index it.
>>> index = similarities.MatrixSimilarity(lsi[another_corpus])
>>>
>>> # Compute similarity of a query vs. indexed documents
>>> sims = index[query]
```

Gensim is a FREE Python library

- ✓ Scalable statistical semantics
- ✓ Analyze plain-text documents for semantic structure
- ✓ Retrieve semantically similar documents


Features

Hover your mouse over each feature for more info.

-  Scalability
-  Platform independent
-  Robust
-  Open source

Scikit-Learn.org NLP Module is not there anymore

[→](#) [↶](#) [🏠](#) https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction [☆](#) [🔍](#)



[Previous](#)
4.1. Pipeline...

[Next](#)
4.3. Preproce...

[Up](#)
4. Dataset tr...

scikit-learn v0.20.3
[Other versions](#)

Please [cite us](#) if you use the software.

4.2. Feature extraction

4.2.1. Loading features from dicts

4.2.2. Feature hashing

- 4.2.2.1. Implementation details

4.2.3. Text feature extraction

- 4.2.3.1. The Bag of Words representation
- 4.2.3.2. Sparsity
- 4.2.3.3. Common Vectorizer usage
 - 4.2.3.3.1. Using stop words
- 4.2.3.4. Tf-idf term weighting
- 4.2.3.5. Decoding text files
- 4.2.3.6. Applications and examples
- 4.2.3.7. Limitations of the Bag of Words representation
- 4.2.3.8. Vectorizing a large text corpus with the hashing trick
- 4.2.3.9. Performing out-of-core scaling with HashingVectorizer
- 4.2.3.10. Customizing the vectorizer classes

4.2.3. Text feature extraction

4.2.3.1. The Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length.

In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely:

- **tokenizing** strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
- **counting** the occurrences of tokens in each document.
- **normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

In this scheme, features and samples are defined as follows:

- each **individual token occurrence frequency** (normalized or not) is treated as a **feature**.
- the vector of all the token frequencies for a given **document** is considered a multivariate **sample**.

A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus.

We call **vectorization** the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag of Words** or “Bag of n-grams” representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

4.2.3.2. Sparsity

As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have many feature values that are zeros (typically more than 99% of them).

Word vectors for 157 languages · x +

fasttext.cc/docs/en/crawl-vectors.html

fii Docs Resources Blog GitHub

Resources

- English word vectors
- [Word vectors for 157 languages](#)
- Wiki word vectors
- Aligned word vectors
- Supervised models
- Language identification
- Datasets

Word vectors for 157 languages

We distribute pre-trained word vectors for 157 languages, trained on *Common Crawl* and *Wikipedia* using fastText. These models were trained using CBOW with position-weights, in dimension 300, with character n-grams of length 5, a window of size 5 and 10 negatives. We also distribute three new word analogy datasets, for French, Hindi and Polish.

Download directly with command line or from python

In order to download with command line or from python code, you must have installed the python package as [described here](#).

Command line Python



















```
$ ./download_model.py en # English
Downloading https://dl.fbaipublicfiles.com/fasttext/vectors-crawl/cc.en.300.bin.gz
(19.78%) [=====> ]
```

Once the download is finished, use the model as usual:

```
$ ./fasttext nn cc.en.300.bin 10
Query word?
```


Spark NLP, John Snow LABS

- <https://www.johnsnowlabs.com/spark-nlp-in-action/>

| | | | |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Spark NLP: English |  Medical Large Language Models Explore the use of Medical Large Language Models for tasks like Text Summarization, Generation, and Question Answering. |  Detect Entities in Clinical Text Identify 77 entity types including Symptom, Treatments, Test, Oncological, Procedure, Diabetes, Drug, Dosage, Date, Imaging (...) |  Information Extraction in Oncology Detect clinical entities and relationships related to cancer staging, grading, histology, tumor characteristics, biomarkers, (...) |
| Spark NLP: World Languages |  Live Demo |  Live Demo |  Live Demo |
| Clinical NLP |  Colab |  Colab |  Colab |
| Voice of Patients |  De-identify Clinical Notes in Different Languages De-identify and obfuscate protected health information (PHI) in English, Spanish, French, Italian, Portuguese, Romanian, and Ger (...) |  Adverse Drug Event Detection Detect adverse reactions from drugs described in the clinical text, online reviews, and social media posts. |  Voice of the Patients Extract and classify healthcare-related terms from documents written by patient such as questions, reviews, messages, and soc (...) |
| Medical NLP: World Languages |  Live Demo |  Live Demo |  Live Demo |
| Medical Large Language Models |  Colab |  Colab |  Colab |

- Spark NLP is an API built by John Snow Labs atop of Apache (Databricks) Spark. This API is not part of Apache Spark API.
- Spark NLP spans the classical and deep learning space.
- John Snow Labs claims to be the best in almost any category of NLP.

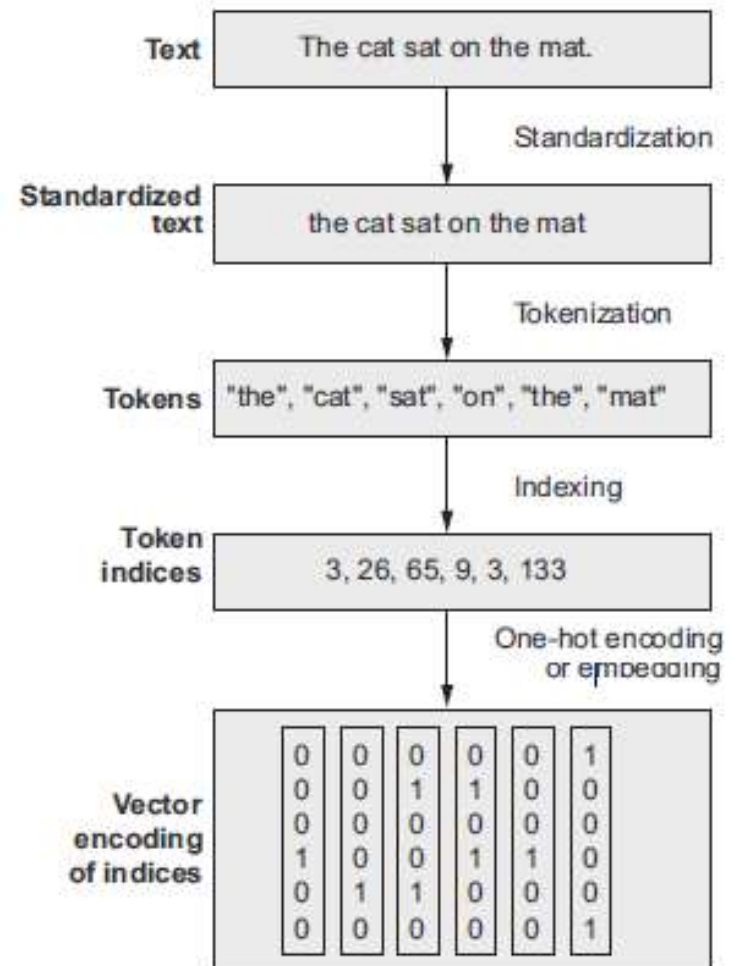
Representations and Text Preparations

Representations of Text

- Historically, we represent text in several ways:
 - In older, but still used approach, we represent the content (sentences, paragraphs, documents) as sets of words, ignoring the order of words. Such approach gives us the "bag-of-words" models.
 - In another approach, we keep the natural order of words. This approach gives us the "sequence models". Tools like RNNs are designed to analyze such models.
 - It is possible to use mixed models, in which we preserve partial information about local order of words. One such approach uses n-grams, rather than the individual words. Those n-grams are eventually tossed into a bag-of-n-grams.
- All models create numerical representation for each word or token.
 - "Classical" models treat each word (token) as a discrete, separate, dimension (axis) in a multi-dimensional vector space and every text has various projections on those word axes.
 - More modern techniques use lower dimensional vector spaces, where axes are important (semantic) concepts, and every word (token) has a distributed representation with projections along many or all semantic axes. Such representations of words are referred to as embeddings or distributed representations. Multiword texts are assemblies of such distributed words.
- Libraries like NLTK (Natural language tool kit), SpaCy, and Gensime, help you tokenize any text so it could be processed as collection of numerical symbols
- Those libraries convert documents into vectors, using `tf-idf`, `word2vec` or other technique.

Preparing, Vectorizing text data

- Deep learning models, being differentiable functions, can only process numeric tensors. They can not take raw text as input. Vectorizing text is the process of transforming text into numeric tensors.
- Text vectorization processes come in many shapes and forms, but they all follow the same template:
 - First, you *standardize* the text to make it easier to process, such as by converting it to lowercase or removing punctuation.
 - You split the text into units (called *tokens*), such as characters, parts of words, words, or groups of words. This is called *tokenization*.
 - You convert each such token into a numerical vector. Usually, this first involves *indexing* all tokens present in the data.



Text standardization

- Consider these two sentences:
 - "sunset came. i was staring at the Mexico sky. Isnt nature splendid??"
 - "Sunset came; I stared at the México sky. Isn't nature splendid?"
- They're very similar. If you were to convert them to byte strings, they would end up with very different representations, because "i" and "I" are two different characters, "Mexico" and "México" are two different words, "isnt" isn't "isn't," and so on. An ML model doesn't know that "staring" and "stared" are two forms of the same verb.
- Text standardization is a basic form of feature engineering that aims to erase encoding differences that you don't want your model to deal with.
- One of the simplest and most widespread standardization schemes is "convert to lowercase and remove punctuation characters." Another common transformation is to convert special characters to a standard form, such as replacing "é" with "e," "æ" with "ae," and so on. A more advanced standardization pattern is stemming: converting variations of a term (such as different conjugated forms of a verb) into a single shared representation, like turning "caught" and "been catching" into "[catch]" or "was staring" and "stared" into "[stare]". Two above sentences are now:
 - "sunset came i [stare] at the mexico sky isnt nature splendid"
- With these standardization techniques, your model will require less training data and will generalize better. Standardization may also erase some information, so always keep the context in mind.

Text Splitting & Text Processing Models

- Once your text is standardized, you need to break it up into units (tokens) in a step called **tokenization**. You could do this in a few different ways:
 - **Word-level tokenization**—Tokens are space-separated (or punctuation separated) substrings.
 - A variant of this is to further **split words into subwords** when applicable—for instance, treating "staring" as "star+ing" or "called" as "call+ed."
 - **N-gram tokenization**—Where tokens are groups of N consecutive words. For instance, "the cat" or "he was" would be 2-gram tokens (also called bigrams).
 - **Character-level tokenization**—Where each character is its own token. In practice, this scheme is rarely used, and you only really see it in specialized contexts, like text generation or speech recognition.
- Most often we use either word-level or N-gram tokenization.
- There are **two kinds of text-processing models**:
 - **sequence model**, that care about word order, and
 - **bag-of-words models**, treating words as a set and disregard their original order.
- If you're building a sequence model, use word-level tokenization,
- If you're building a bag-of-words model, use N-gram tokenization. N-grams are a way to artificially inject a small amount of local word order information into the model.

Vocabulary Indexing

- Once your text is split into tokens, you need to encode each token into a numerical representation. You could potentially do this in a stateless way, such as by hashing each token into a fixed binary vector, but in practice, we build an index of all terms found in the training data (the "**vocabulary**") and assign a unique integer to each entry in the vocabulary. Something like this:

```
vocabulary = {}  
for text in dataset:  
    text = standardize(text)  
    tokens = tokenize(text)  
    for token in tokens:  
        if token not in vocabulary:  
            vocabulary[token] = len(vocabulary)
```

- You could convert those integers into vector encodings that can be processed by a neural network, like **one-hot vector encodings**:

```
def one_hot_encode_token(token):  
    vector = np.zeros((len(vocabulary),))  
    token_index = vocabulary[token]  
    vector[token_index] = 1  
    return vector
```

- It is common to restrict the vocabulary to only the top 20,000 or 30,000 most common words found in the training data. Indexing very rare terms would result in an excessively large feature space, where most features would have little information content.

OOV Index, Padding

- When we look up a new token in our vocabulary index, it may not necessarily exist. Your training data may not have contained any instance of the word "cherimoya" (or maybe you excluded it from your index because it was too rare), so doing `token_index = vocabulary["cherimoya"]` may result in a `KeyError`.
- To handle this, you should use an "out of vocabulary" index (abbreviated as OOV index)—a catch-all for any token that is not in the index. It's usually index 1. You are actually doing `token_index = vocabulary.get(token, 1)`.
- When decoding a sequence of integers back into words, you replace 1 with something like "[UNK]" (which we call an "OOV token").
- Besides the OOV token (index 1), we also need a mask (padding) token (index 0).
- OOV token means "here was a word we did not recognize".
- The mask token tells us "ignore me, I'm not a word."
- We use the mask token to pad sequence data. Data batches need to be contiguous, and all sequences in a batch of sequence data must have the same length, so shorter sequences should be padded to the length of the longest sequence.
- A batch of data including 2 sequences `[5, 7, 124, 4, 89]` and `[8, 34, 21]`, with masks would look like this:

```
[[5, 7, 124, 4, 89]  
[8, 34, 21, 0, 0]]
```

TextVectorization Layer

- All described steps could readily be implemented by the layer/class `keras.layers.TextVectorization`
- For example, to configure a layer to return sequences of words encoded as integer indices, we write:

```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization(
    output_mode="int",
)
```

- By default, the `TextVectorization` layer will use the settings "[convert to lowercase and remove punctuation](#)" for text standardization, and "[split on whitespace](#)" for tokenization. The layer can handle any use case and we can provide custom functions for standardization and tokenization, if we need to.
- To index the vocabulary of a text corpus, just call the `adapt()` method of the layer with a `Dataset` object that yields strings, or just with a list of Python strings:

```
dataset = ["I write,erase,rewrite","Erase again,and then", "Poppy blooms.",]
text_vectorization.adapt(dataset)
```

- To display the vocabulary, type:

```
>>>text_vectorization.get_vocabulary()
["", "[UNK]", "erase", "write", ...]
```

Encode and Decode a Sentence

- To encode and then decode an example sentence, using the above vocabulary, we type:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7 3 5 9 1 5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in
encoded_sentence)
>>> print(decoded_sentence)
"I write rewrite and [UNK] rewrite again"
```

Representing Text as a Set of Words

Vector Space Model

Vector Space Model or Bag of Words Model

- Classical language processing techniques including the original Google' Page Rank system treated documents $\{d_j\}$ and queries $\{q\}$ as vectors in a very high dimensional vector space

$$d_j = (w_{1,j}, w_{2,j}, \dots, w_{t,j}) , \quad q_k = (w_{1,k}, w_{2,k}, \dots, w_{n,k})$$

- d_j represents a document with index j . t is the dimension of the vector space, i.e., the number of distinct terms in the document or document corpus (collection of documents) or the number of words in the dictionary (vocabulary).
- q_k is a query string. Query string q_k has n distinct terms (words of significance). Each dimension corresponds to a distinct term (word). Index k labels one of query strings, if there are a few of them.
- There are different ways of computing $w_{i,j}$, the **term weights**:
 - In one approach, value of $w_{k,j}$, for the document with index k , is the number of occurrence of token (word) with index j in the corpus of text.
 - A more sophisticated schema is known as **tf-idf** weighting. The acronym **tf-idf** stands for **term frequency-inverse document frequency**.
- The definition of the "term" depends on the application.
- Typically, terms are single words, keywords, or longer phrases.
- If words are chosen to be the terms, the dimensionality of space is the number of words in the vocabulary, i.e. the number of distinct words (terms) occurring in the corpus.
- If ordering of words is considered unimportant, the model is called the **Bag of Words Model**.

tf-idf weights

- In the classical vector space model proposed by Salton, Wong and Yang in 1975 (<https://dl.acm.org/doi/10.1145/361219.361220>) the term specific weights in the document vectors are presented as the products of local and global parameters. The model is known as the *term frequency-inverse document frequency* model.
- Weights $w_{t,d}$ in the vector \mathbf{d}_j representing document j , $\mathbf{d}_j = (w_{1,j}, w_{2,j}, \dots, w_{t,j})$ are calculated as:

$$w_{t,d} = tf_{t,d} \cdot \log \frac{|D|}{|\{d' \in D | t \in d'\}|}$$

- $tf_{t,d}$ is the term frequency, i.e., the number of times term t appears in document d (a local parameter).
- $\log \frac{|D|}{|\{d' \in D | t \in d'\}|}$ is called inverse document frequency.
- $|D|$ is the total number of documents in the document set (corpus).
- $|\{d' \in D | t \in d'\}|$ is the number of documents containing term t . Notice that if a term is present in a smaller number of documents d' , its weight is greater (denominator in the logarithmic expression is smaller). The term with a smaller number of occurrences has a higher discriminatory power.
- This model, while it might appear too simplistic, has been extensively used and is not unsuccessful. Most of the historic *information retrieval applications* used this model.
- Various API-s, Keras, TF, PyTorch provide classes or functions to calculate weights of this model.

Comparing Documents and Queries

- Whether a document is "relevant" as an answer to a query string can be determined in a variety of ways. On the basic level, we usually look for the similarity of the query string and the document string.
- Since both the query and the documents are "vectors", similarity or alignment of those two vectors could conveniently be expressed by the cosine of the angle between two vectors, thus the *cosine similarity*.

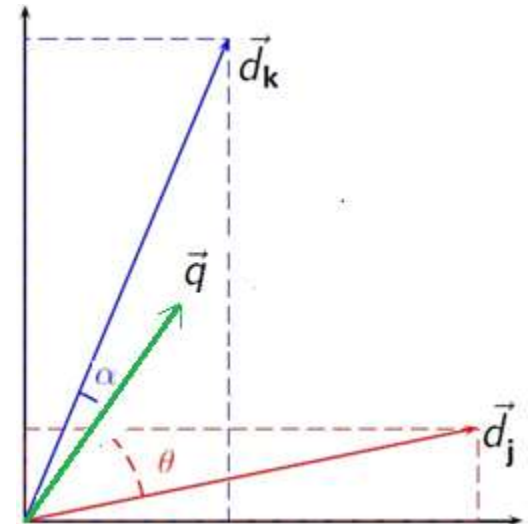
$$\cos(\theta) = \frac{d_j \cdot q}{\|d_j\| \|q\|}$$

- The dot product of two vectors is calculated as:

$$d_j \cdot q = \sum_{m=1}^t w_{m,j} w_{m,q}$$

- In the diagram on the right, vector q is more similar to vector d_k than to the vector d_j , since angle α is smaller than angle θ , i. e. $\cos(\alpha) > \cos(\theta)$
- Norms of vectors are standard Euclidian norms

$$\|q\| = \sqrt{\sum_{i=1}^n q_i^2}$$



Advantages and Limitations of Vector Space Model

- Representation of words and documents using **tf-idf** weighting or similar schema is referred to as the vector space model. The model has some advantages:
 1. The model is simple and based on linear algebra
 2. Term weights are not binary
 3. Model allows computing a continuous degree of similarity between queries and documents
 4. Allows ranking documents according to their possible relevance
 5. Allows partial matching
- The model has a few limitations
 1. High dimensionality determined by the large size of the vocabulary of most languages makes **documents very sparse**.
 2. **Long documents have poor similarity values** with most queries (a small scalar product and a large dimensionality)
 3. **Search keywords must precisely match document terms**; word substrings are not used since they might result in a "false positive match"
 4. Semantic sensitivity; documents with similar context but different term vocabulary are not associated. **Synonyms have zero similarity**.
 5. The **order in which the terms appear in the document is lost** in the vector space representation.
 6. All terms are statistically independent.
 7. Weighting is intuitive but very formal.

Distributional Semantics Models

Foundation of Semantics Models

- The distributional semantics is based on the so-called distributional hypothesis:
"linguistic items with similar distributions (in text) have similar meanings" (1).
- Another way to express this hypothesis is to state:
"words that are used and occur in the same contexts tend to transmit similar meanings." or
"a word is characterized by the company it keeps"
- In recent years, the distributional hypothesis has provided the basis for the theory of similarity-based generalization in language learning. We assume that children can figure out how to use words they rarely encountered before by generalizing about their use from the distributions of similar words.
- The distributional hypothesis suggests that two words more similar in meaning (semantically similar) will tend to occur in similar linguistic contexts.
- This assertion has significant implications for both (the data sparsity problem in) computational modeling, and for the question of how children learn language so rapidly given relatively impoverished input.
- The key concept in the above reasoning is *the linguistic context* (i.e., surrounding of a word in a sentence.)

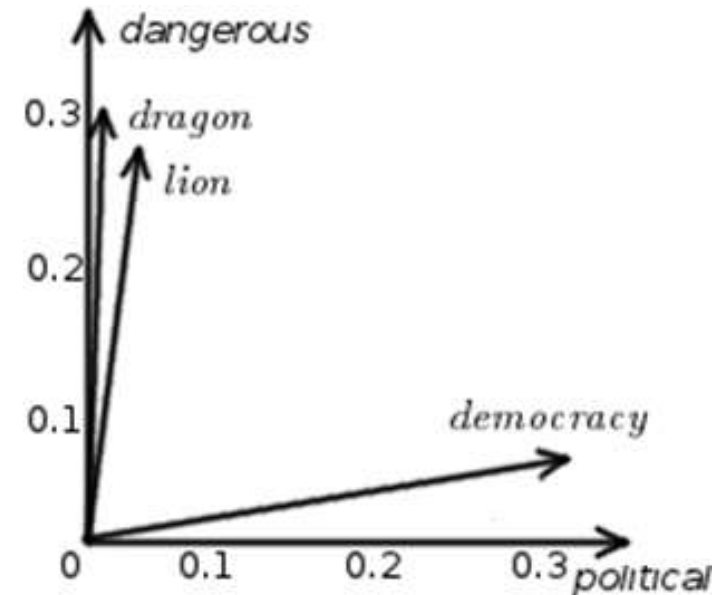
(1) Lenci, Alessandro; Sahlgren, Magnus (2023). *Distributional Semantics*. Cambridge University Press.

Linguistic Context

- The linguistic context of a word is the collection of other words that appear next to it. For example, the following might be a context for 'coconut':
a fruit found throughout the tropic and subtropic area, the . . coconut. . . is known for its great versatility
- If we identify every single instance of the word 'coconut' in Wikipedia, and start counting how many times it appears next to 'tropic', 'versatility', 'the', 'subatomic', etc., we find that coconuts appear many more times next to 'tropic' than next to 'subatomic'.
- But counting is not enough. The word 'coconut' appears thousands of times next to 'the'. 'the' tells us little about what coconuts are.
- Usually, the raw counts are combined into statistical measures which help us define more accurately which words are 'characteristic' for a particular term. *Pointwise mutual information* is one such measure, which gives higher values to words that are really defining for a term: for instance, 'tropic', 'food', 'palm' for 'coconut'.
- Once we have such measures, we can build a simulation of how the words in a language relate to each other. This is done in a so-called '*semantic space*' which, mathematically, corresponds to a vector space. The dimensions of the vector space are context words like 'tropic' or 'versatility' and words are defined as points (or vectors) in that space.
- *Semantic space* is of a lower dimension than previously discussed *Term Vector Space*.

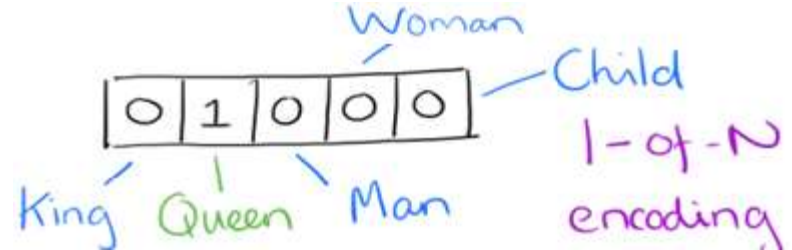
Similarity in Semantic Space

- Here is a very simplified example, where we define words '*dragon*', '*lion*' and '*democracy*' with respect to only two dimensions: '*dangerous*' and '*political*'.
- Because dragons and lions are very dangerous but have very little with political systems, their vectors have a high value along the 'dangerous' axis and a small value along the 'political' axis.
- The opposite is true for 'democracy'.
- Note that very naturally, the vectors for 'dragon' and 'lion' have clustered together in the space, while 'democracy' is much further from them.
- By calculating the (angular) distance between two vectors in a semantic space, we can deduce the similarity of two words: this is what allows us to say that cats and dogs are more alike than cats and coconuts.
- In regular usage, the dimensionalities of semantic spaces are several hundreds and at most several thousands, but never hundreds of thousands or millions.



One-hot Encoding vs. Semantic Vector Space

- In a simple 1-of-N (or 'one-hot') encoding every element in the vector is associated with a word in the vocabulary. The encoding of a given word is simply the vector in which the one element is set to one, and all other elements are zero.
- Suppose our vocabulary has only five words:
- King, Queen, Man, Woman, and Child
- We could encode the word 'Queen' as



0 1 0 0 0. Normal vocabulary has tens of thousand words

- **Word2vec learns "distributed" representation of words.** Such vector spaces usually have several hundred dimensions (say 200 or 1000). Each word is represented by a distribution of weights across those dimensions. So, instead of a one-to-one mapping between an element in the vector and a word, the representation of a word is spread across all the elements in the vector, and each element in the vector contributes to the definition of many words.
- If we label the dimensions in a hypothetical space as: Royalty, Masculinity, Femininity, Age, ... (there are no such pre-assigned labels in the algorithm) above vectors might look like these:
- In an abstract way, such vectors represent the 'meaning' of words. By examining a large corpus it is possible to learn word vectors that capture such **semantic or distributed relationships between words.**

A diagram showing four vertical vectors for the words 'King', 'Queen', 'Woman', and 'Princess'. To the left, four dimensions are listed: 'Royalty', 'Masculinity', 'Femininity', and 'Age'. Each vector contains numerical values for these dimensions.

| | King | Queen | Woman | Princess |
|-------------|------|-------|-------|----------|
| Royalty | 0.99 | 0.99 | 0.02 | 0.98 |
| Masculinity | 0.99 | 0.05 | 0.01 | 0.02 |
| Femininity | 0.05 | 0.93 | 0.999 | 0.94 |
| Age | 0.7 | 0.6 | 0.5 | 0.1 |

Applications of Distributional Semantics

- Semantics is the study of meaning. Semantics investigates questions such as:
 - What is meaning?
 - Why words and sentences have meaning?
- **Distributional semantic models** were successfully applied in many use cases:
 - language translation
 - finding similarity between words and multiword expressions;
 - word clustering based on similarity;
 - automatic creation of thesauri and bilingual dictionaries;
 - lexical ambiguity resolution;
 - expanding search requests using synonyms and associations;
 - defining the topic of a document;
 - document clustering for information retrieval;
 - data mining and named entities recognition;
 - creating semantic maps of different subject domains;
 - paraphrasing;
 - sentiment analysis;
 - modeling preferences of words.

Word2Vec Embedding Spaces

Word2Vec

- Google's researchers (Tomas Mikolov et al.) created a tool called **Word2Vec** which efficiently implements ideas of Distributed Semantics.
- Word2Vec is a neural network that processes text. Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus.
- The original Word2vec was a shallow neural network. However, it turned text into a numerical format that deep neural networks could understand.
- The **Word2Vec** tool first extracts a vocabulary from the training text data and then learns vector representation of words. The resulting **word vectors are called embeddings** and are used as features in many natural language processing and machine learning applications.
- With Word2Vec learned representations (embeddings) and **cosine distance** we could find the closest words for any user-specified word.
- For example, for word 'france', **cosine distance** will display the most similar words and their distances to 'france', which should look like:

Word Cosine distance

| | |
|-------------|----------|
| spain | 0.678515 |
| belgium | 0.665923 |
| netherlands | 0.652428 |
| italy | 0.633130 |
| switzerland | 0.622323 |

Word Cosine distance

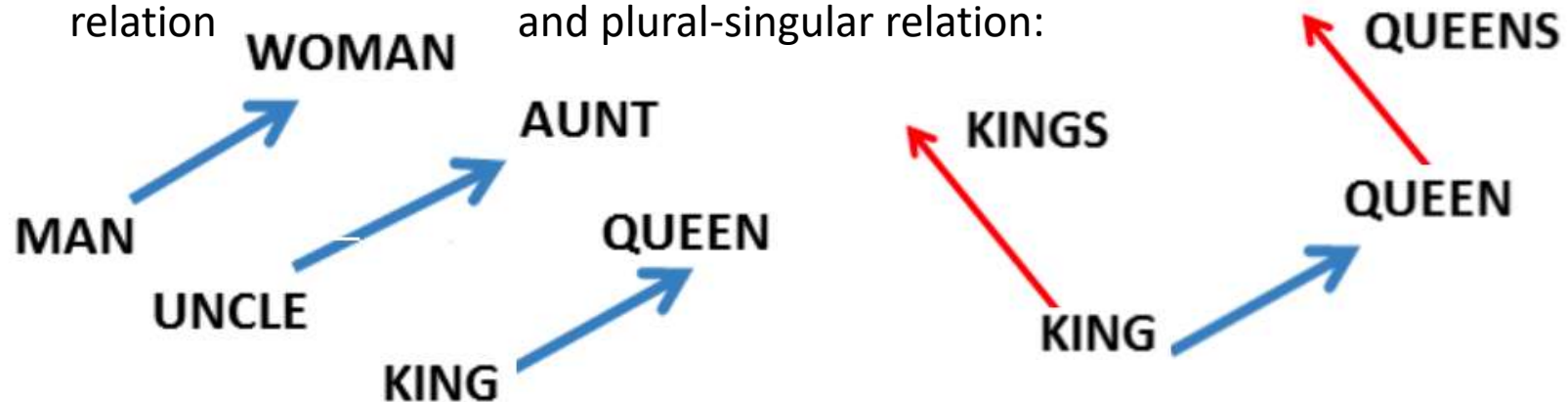
| | |
|------------|----------|
| luxembourg | 0.610033 |
| portugal | 0.577154 |
| russia | 0.571507 |
| germany | 0.563291 |
| catalonia | 0.534176 |

Reasoning with word vectors

- We found that word representations learned by **Word2vec** capture meaningful syntactic and semantic regularities in a very simple way.
- Specifically, the regularities are observed as constant vector offsets between pairs of words sharing a particular relationship.
- For example, if we denote the vector for word i as x_i , and focus on the singular/plural relation, we observe that

$$x_{apple} - x_{apples} \approx x_{car} - x_{cars}, \quad x_{family} - x_{families} \approx x_{cat} - x_{cats}, \text{ and so on.}$$

- This appears to be the case for a variety of semantic relations.
- The distributed vectors are very good at answering analogy questions of the form: *a is to b as c is to ?* For example, *man is to woman as uncle is to ? (aunt)*.
- When measuring the similarity, we use the cosine similarity.
- For example, below are vector offsets for three word pairs illustrating the gender relation and plural-singular relation:



“Algebraic” Operations on Vectors

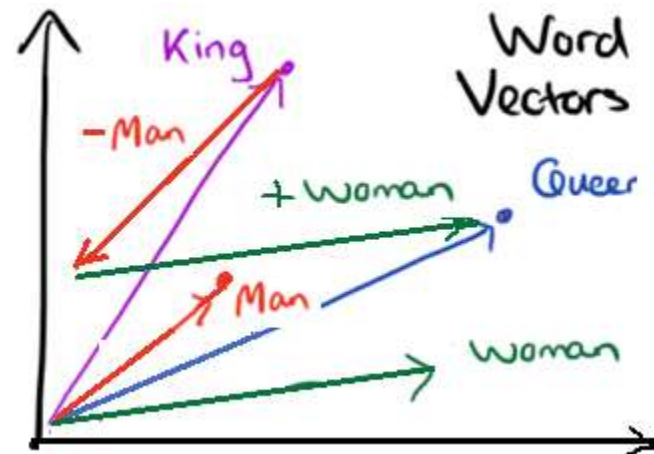
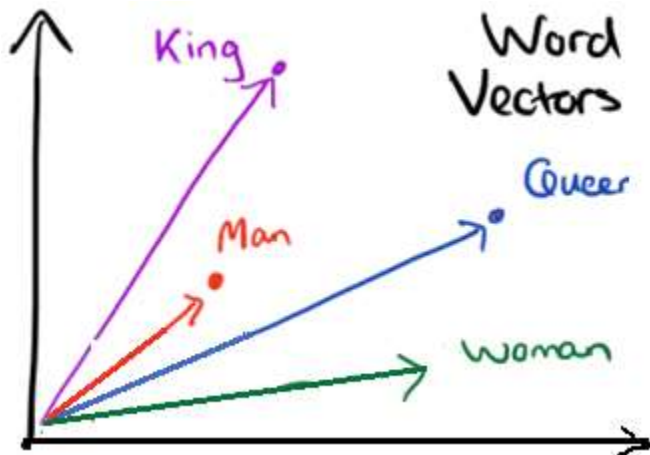
- It appears that we could perform "vector algebra" on "words" and calculate expressions like:

$$\text{vector}(\textit{King}) - \text{vector}(\textit{Man}) + \text{vector}(\textit{Woman}) \sim \text{vector}(\textit{Queen})$$

- The result will be a vector that is very close to the vector representation of the word "Queen".
- All of this is truly remarkable. All this **knowledge** simply comes from looking at many words in context with no other information provided about their semantics.
- It was found that similarity of word representations goes beyond simple syntactic regularities.

Vector Subtraction and Addition

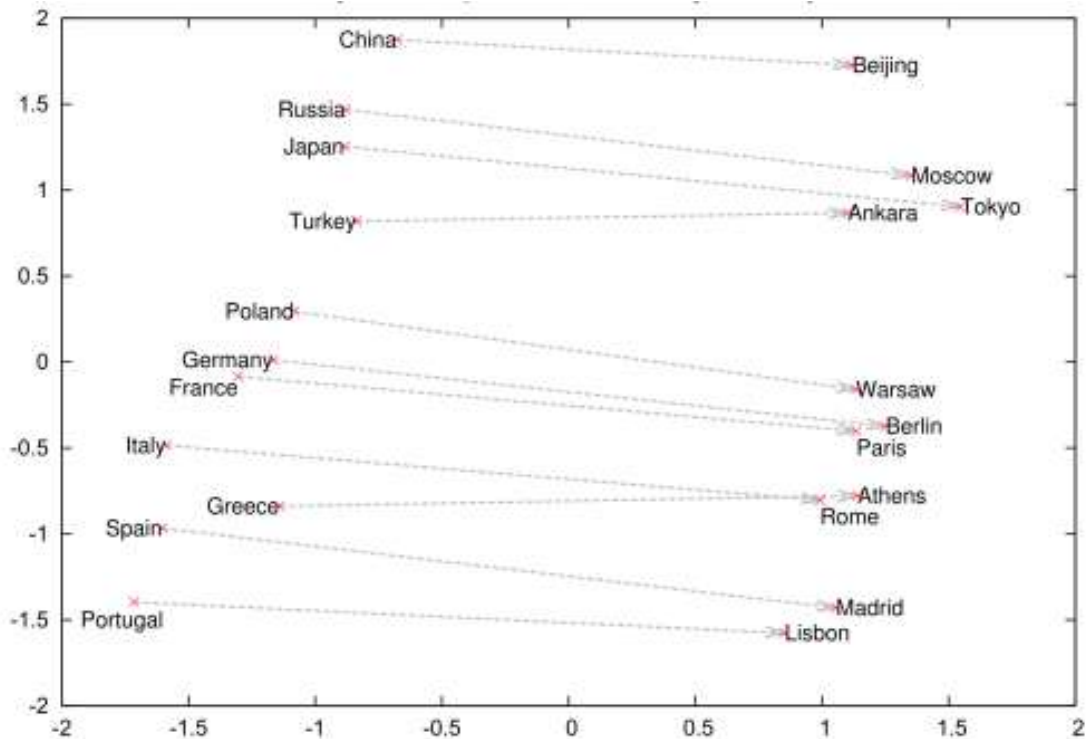
- If you recall vector operations, you remember that you can add one vector to another.
- Similarly, you can subtract one vector from another.
- Subtraction is the same as adding a negative vector.
- So, if we subtract the vector for word "man" from the vector for word "king" and then add to the resulting vector a vector for word "woman", we arrive at the vector for word "queen".
- Those operations are schematically presented bellow:



$$Queen = King + (-Man) + Woman$$

Countries vs. Capitals, PCA 2-d Projection

- Below is a 2-dimensional (PCA) projection of the 1000-dimensional Skip-gram vectors of countries and their capitals. This illustrates the ability of the model to automatically organize concepts and learn implicitly the relationships between them. During the training, no supervised information was provided suggesting what is the meaning of the words representing capital cities or countries.
- For some reasons, Turkey and Ankara are connected by a flat vector different from most other country-capital pairs.
- The same or some other reasons explain the same flat vector between Greece and Athens.
- Greece and Turkey are neighbors in a quarrel without an end?



Examples of Word-Pair Relationships

- Examples of the word pair relationships using the best word vectors (skip-gram model trained on 783 M words in a space of 300 dimensions).
- For example, if you say relationship is France – Paris, what is the corresponding word for Italy, the answer will be Rome.
- Similarly, if the basic relationship is Japan – sushi, and we ask for the word corresponding to Germany we will get bratwurst.

| Relationship | Example 1 | Example 2 | Example 3 |
|----------------------|---------------------|-------------------|----------------------|
| France - Paris | Italy: Rome | Japan: Tokyo | Florida: Tallahassee |
| big - bigger | small: larger | cold: colder | quick: quicker |
| Miami - Florida | Baltimore: Maryland | Dallas: Texas | Kona: Hawaii |
| Einstein - scientist | Messi: midfielder | Mozart: violinist | Picasso: painter |
| Sarkozy - France | Berlusconi: Italy | Merkel: Germany | Koizumi: Japan |
| copper - Cu | zinc: Zn | gold: Au | uranium: plutonium |
| Berlusconi - Silvio | Sarkozy: Nicolas | Putin: Medvedev | Obama: Barack |
| Microsoft - Windows | Google: Android | IBM: Linux | Apple: iPhone |
| Microsoft - Ballmer | Google: Yahoo | IBM: McNealy | Apple: Jobs |
| Japan - sushi | Germany: bratwurst | France: tapas | USA: pizza |

Analogical Reasoning

- Here are some more examples of the 'a is to b as c is to ?' style questions answered by word vectors. The goal is to compute the fourth phrase using the first three.
- The authors of the original paper claim that they could achieve 72% accuracy on a test set of 3218 examples.

| Newspapers | | | |
|--------------------------------------|------------------------------------------|-----------------------------|-------------------------------------------|
| New York San Jose | New York Times San Jose Mercury News | Baltimore Cincinnati | Baltimore Sun Cincinnati Enquirer |
| NHL Teams | | | |
| Boston Phoenix | Boston Bruins Phoenix Coyotes | Montreal Nashville | Montreal Canadiens Nashville Predators |
| NBA Teams | | | |
| Detroit Oakland | Detroit Pistons Golden State Warriors | Toronto Memphis | Toronto Raptors Memphis Grizzlies |
| Airlines | | | |
| Austria Belgium | Austrian Airlines Brussels Airlines | Spain Greece | Spainair Aegean Airlines |
| Company executives | | | |
| Steve Ballmer Samuel J. Palmisano | Microsoft IBM | Larry Page Werner Vogels | Google Amazon |

Linguistic Regularity in Word Vector Space

| <i>Expression</i> | <i>Nearest token</i> |
|-----------------------------------------|----------------------|
| Paris - France + Italy | Rome |
| bigger - big + cold | colder |
| sushi - Japan + Germany | bratwurst |
| Cu - copper + gold | Au |
| Windows - Microsoft + Google | Android |
| Montreal Canadiens - Montreal + Toronto | Toronto Maple Leafs |

Element-wise Addition of vector elements

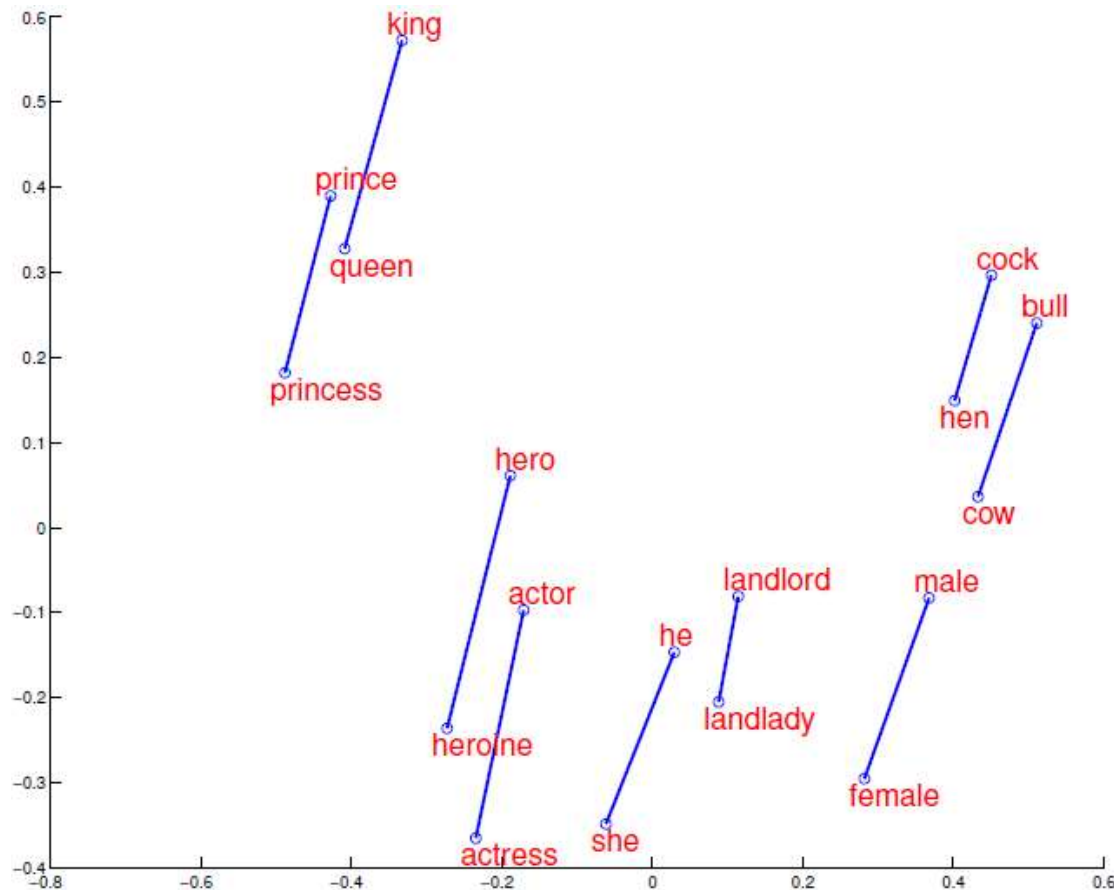
- We can also use element-wise addition of vector elements to ask questions such as 'German + airlines' and by looking at the closest tokens to the composite vector produce impressive answers:

| Czech + currency | Vietnam + capital | German + airlines | Russian + river | French + actress |
|------------------|-------------------|------------------------|-----------------|----------------------|
| koruna | Hanoi | airline Lufthansa | Moscow | Juliette Binoche |
| Check crown | Ho Chi Minh City | carrier Lufthansa | Volga River | Vanessa Paradis |
| Polish zolty | Viet Nam | flag carrier Lufthansa | upriver | Charlotte Gainsbourg |
| CTK | Vietnamese | Lufthansa | Russia | Cecile De |

- Vector compositionality using element-wise addition. Four closest tokens to the sum of two vectors are shown, using the best Skip-gram model.
- Word vectors with such semantic relationships could be used to improve many existing NLP applications, such as machine translation, information retrieval and question answering systems, and may enable other future applications yet to be invented.

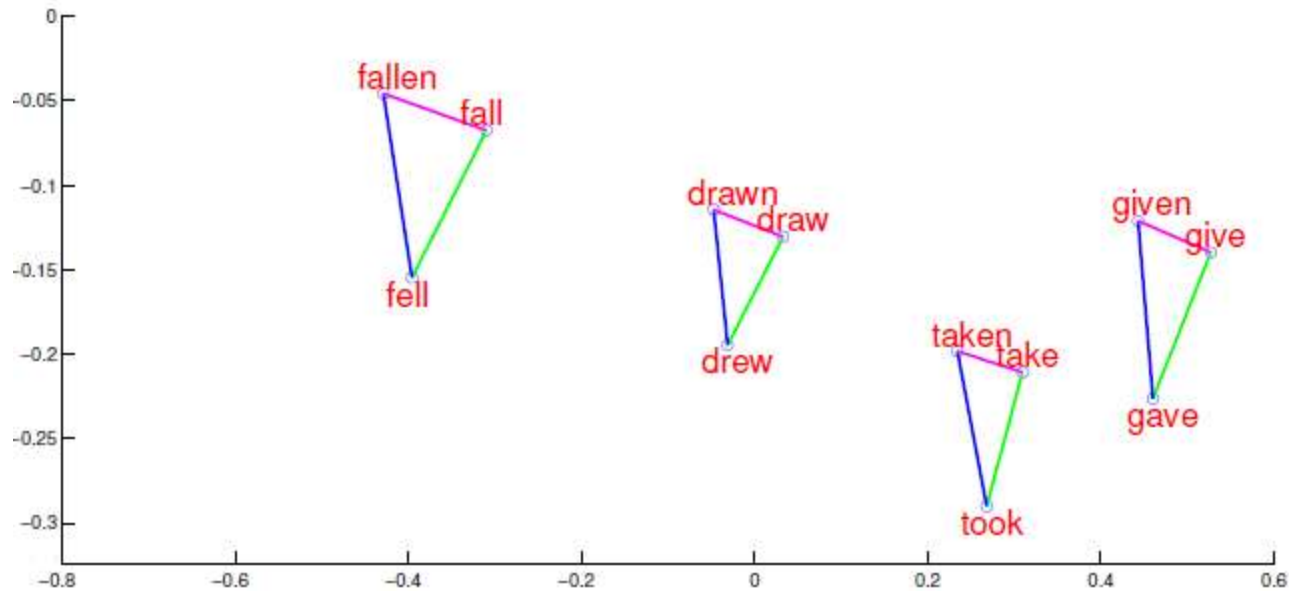
Visualization of Regularities in Word Vector Space

- We can visualize the word vectors by projecting them to 2D space
- PCA can be used for dimensionality reduction
- Although a lot of information is lost, the regular structure is often visible
- Masculine-feminine relationship is well preserved



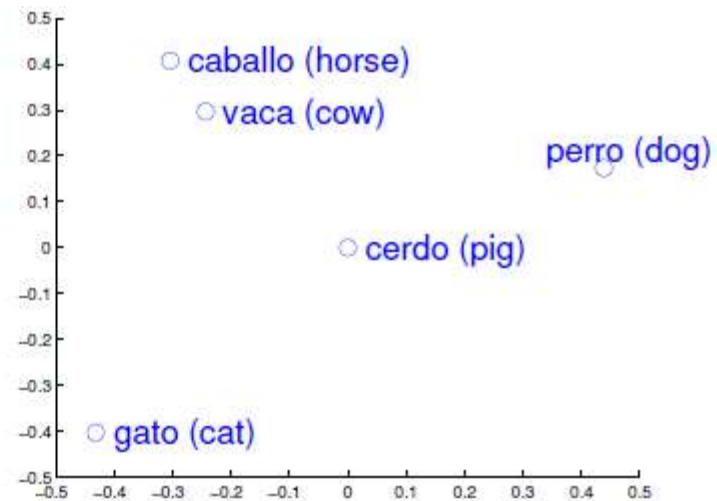
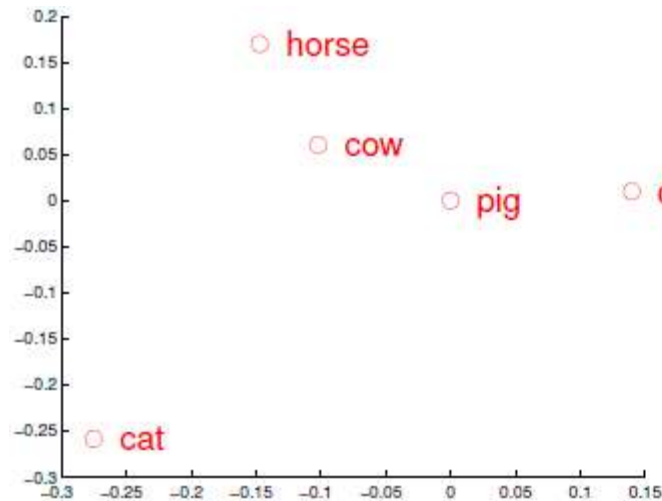
Visualization of Regularity in Word Vector Spaces

- Relationships between verb tenses are well maintained.



Machine Translation

- Word vectors should have similar structure when trained on comparable corpora
- This should hold even for corpora in different languages



- The figures were manually rotated and scaled.
- For translation from one vector space to another, we need to learn a linear projection (will perform rotation and scaling)
- Small starting dictionary can be used to train the linear projection
- Then, we can translate any word that was seen in the monolingual data

How word2vec Operates

What word2vec does

- The original implementation of word2vec did the following:
 1. Created a 3 layer neural network. (1 input layer + 1 hidden layer + 1 output layer)
 2. Feed the network with a word and train it to predict its neighboring words.
 3. Remove the last (output layer) and keep the input and the hidden layer.
 4. Now, input a word from within the vocabulary. The output given at the hidden layer is the '*word embedding*' of the input word.
- Word2Vec is training a simple neural network with a single hidden layer to perform task 2.
- Then, we do not use that whole neural network.
- Instead, just learn and preserve the weights of the hidden layer.
- The hidden layer is just like the codings layer or the latent space of the auto-encoder.
- Every word in a text corpora gets one (embedded) vector in that latent space.

Learning Word Vectors

- Tomas Mikolov et al. were not the first to use continuous vector representations of words, but they did reduce the computational complexity of learning such representations – making it practical to learn high dimensional word vectors on a large amount of data.
- For example, they used a Google News corpus for training the word vectors. This corpus contains about 6B tokens. They restricted the vocabulary to the 1 million most frequent words.
- The complexity in neural network language models (feedforward or recurrent) comes from the non-linear hidden layer(s).
- Google's team introduced 2 architectures for calculation of word vectors:
 - *Continuous Bag-of-Words* model, and a
 - *Continuous Skip-gram* model
- Those architectures gave similar but not identical results.

Continuous Bag-of-Words (CBOW) Model

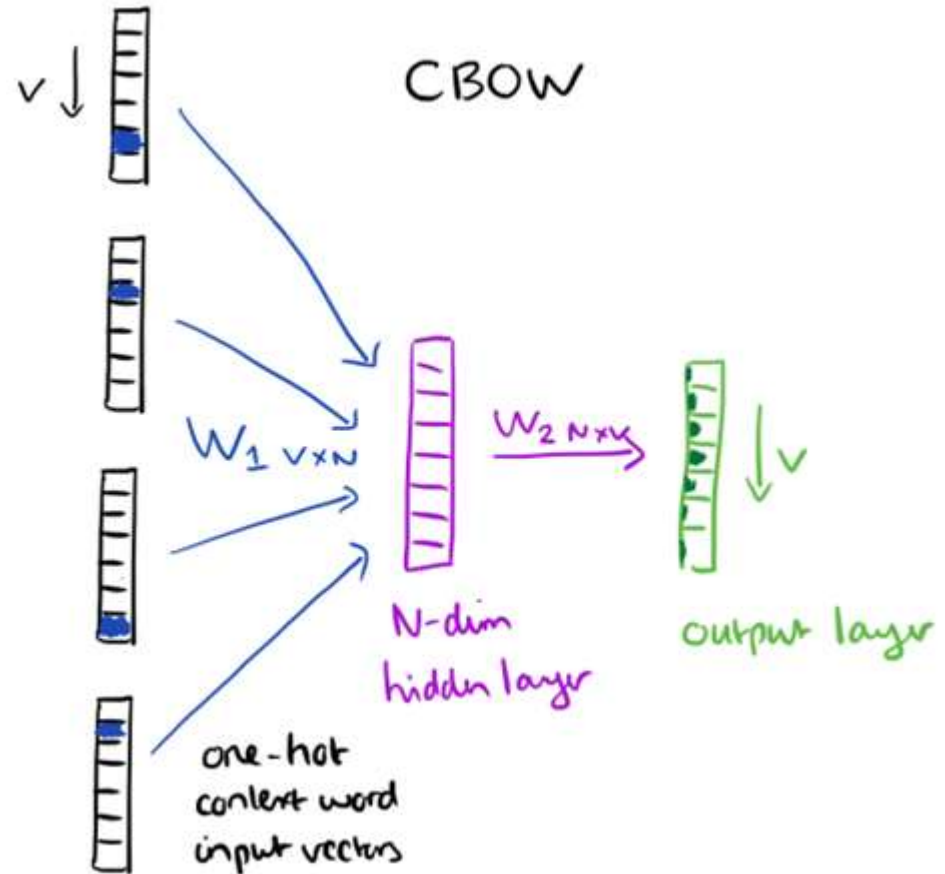
- Consider a piece of text such as "The recently introduced continuous Skip-gram model is an efficient method for learning high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships."
- Imagine a sliding window over the text, that includes the central word currently in focus, together with the four words that precede it, and four words that follow it. Those are the context words.

*an efficient method for **learning** high-quality distributed vector*
*----- context ----- **focus word** ----- context -----*

- The context words form the input layer. Each word is encoded in one-hot form, so if the vocabulary size is V , words are represented by V -dimensional vectors with just one of the elements set to one, and the rest all zeros.
- The network is trained to predict the focus word based on the context.
- The neural network used in the original work has a single hidden layer and an output layer.

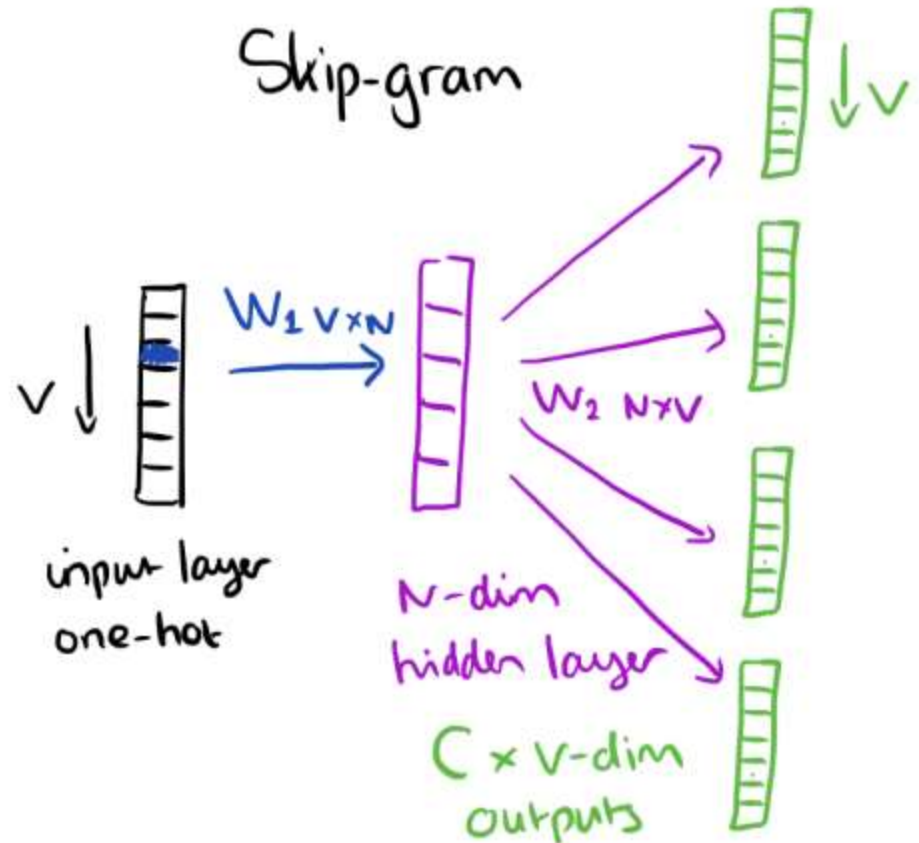
CBOW

- The training objective is to maximize the conditional probability of observing the actual output word (the focus word) given the input context words, regarding the weights.
- In our example, given the input ("an", "efficient", "method", "for", "high", "quality", "distributed", "vector") we want to maximize the probability of getting "learning" as the output.



Skip-gram Model

- The **skip-gram** model is somewhat different. It is constructed with the focus word as the single input vector, and **C** words in the context are now the targets at the output layer:
- The activation function for the hidden layer simply amounts to copying the corresponding row from the weights matrix \mathbf{W}_1 (linear).
- At the output layer, we now output C multinomial distributions instead of just one.
- The training objective is to minimize the summed prediction error across all context words in the output layer. In our example, the input would be "learning", and we hope to see ("an", "efficient", "method", "for", "high", "quality", "distributed", "vector") at the output layer



Installing Word2Vec

Installing Word2Vec

- The original Word2Vec code could be downloaded from the GitHub:
<https://github.com/tmikolov/word2vec>
- Once you expand the archive, you will see a bunch of C language source files: `word2vec.c`, `word2phrase.c`, `word-analogy.co`, `distance.c` and `compute-accuracy.c` and a bunch of demo `sh` scripts that will run various examples.
- C code needs to be compiled before you could use its executables.
- I tried using Windows' Visual Studio `nmake` to compile the code but gave up and used `gcc` compiler and the `make` that I installed with Cygwin. When upgrading Cygwin, please include `wget` as well. If on Windows, instead of Cygwin you could use Windows Subsystem for Linux (WSL) version 2.
- On the Cygwin's or WSL command prompt, in the directory with C files and the `makefile`, type:

```
$ make
```
- You might see a few warnings. Nothing important. If you have them, please remove line `"char ch;"` in two or three files and all warnings will go away.

Installing gcc on Windows

- I use Cygwin for various Linux commands. You may enable Windows Subsystem for Linux (<https://docs.microsoft.com/en-us/windows/wsl/install-win10>)
- You fetch Cygwin installation script: `setup-x86_64.exe` from Cygwin.com.
- Subsequently, as an administrator, on Windows prompt run the following, all on one line:

```
C:\> setup-x86_64.exe -q -P wget -P gcc-g++ -P make -P diffutils -P libmpfr-devel -P libgmp-devel -P libmpc-devel
```

- This installs `gcc` compiler and `wget` among other things.
- You can use graphical installation tool as well. Pay attention to add all above libraries.
- If you are installing Cygwin, and have not done it already, please go to `net(work)` packages and install `openssh`.
- Once it is installed, open Cygwin prompt, and type:

```
$ gcc --version
gcc (GCC) 10.2.0
Copyright (C) 2020 Free Software
Foundation, Inc.
```



Options for Installing `make`

- **Windows:**
 - Use Cygwin to install `make` and `wget`
 - You can use Windows System for Linux as well.
 - <https://osdn.net/projects/mingw/downloads/68260/mingw-get-setup.exe/>
- **Macintosh:**
 - <https://stackoverflow.com/questions/10265742/how-to-install-make-and-gcc-on-a-mac>
- **Linux and WSL:** tools we need: `gcc`, `make`, `wget`, `openssh`. These tools are already built-in in Linux operating systems such as CentOS, RHEL and other Linux variants.

Windows subsystem for Linux

- If your operating system is Windows, you might want to install WSL follow these instructions:

<https://docs.microsoft.com/en-us/windows/wsl/install>

- Or these

<https://www.windowscentral.com/install-windows-subsystem-linux-windows-10>

A Note: Replace `malloc.h` with `stdlib.h`

- When trying to compile older version of word2vec code you might discover that C code is referencing an outdated library called `malloc.h`.
- You just need to update the C code to point to the right library, now called `stdlib.h`.
- Download `word2vec.zip` and unzip it
- Open up each `.c` file and replace each `#include <malloc.h>` with `#include <stdlib.h>`
- Version of word2vec that can be found on the GitHub, does not have this issue.

makefile

- **File** `makefile` which controls the compilation process reads:

```
CC = gcc
#Using -Ofast instead of -O3 might result in faster code, but is
#supported only by newer GCC versions
CFLAGS = -lm -pthread -O3 -march=native -Wall -funroll-loops -Wno-unused-result
all: word2vec word2phrase distance word-analogy compute-accuracy
word2vec : word2vec.c
    $(CC) word2vec.c -o word2vec $(CFLAGS)
word2phrase : word2phrase.c
    $(CC) word2phrase.c -o word2phrase $(CFLAGS)
distance : distance.c
    $(CC) distance.c -o distance $(CFLAGS)
word-analogy : word-analogy.c
    $(CC) word-analogy.c -o word-analogy $(CFLAGS)
compute-accuracy : compute-accuracy.c
    $(CC) compute-accuracy.c -o compute-accuracy $(CFLAGS)
    chmod +x *.sh
clean:
    rm -rf word2vec word2phrase distance word-analogy compute-accuracy
```

demo-word.sh Linux Script and makefile

- On Linux (Cygwin) prompt open and examine `demo-word.sh` script.

```
make
```

```
if [ ! -e text8 ]; then
```

```
    wget http://matthmahoney.net/dc/text8.zip -O text8.gz
```

```
    gzip -d text8.gz -f
```

```
fi
```

```
time ./word2vec -train text8 -output vectors.bin -cbow 1 -size 200 -window 8 -
```

```
negative 25 -hs 0 -sample 1e-4 -threads 20 -binary 1 -iter 15
```

```
./distance vectors.bin
```

- Invoke the script on Cygwin (Linux) command prompt, by typing:

```
$ ./demo-word.sh      # Note ./ is important. Your script is not in your PATH
```

- The script runs `make`, if not run already.
- The script is testing whether file `text8.zip` exists locally. If not, it downloads it using `wget` utility from <http://matthmahoney.net/dc/text8.zip> as a `gz` archive. Subsequently, script expands `gz` archive into file `text8`. The text file `text8` contains a small (97.7 MB) text corpus from the web. `text8` file contains standardized text with 0 lines, 17 million words and 100 million characters Neural network inside `word2vec` trains a small word vector model.
- Switch `-binary 0` should result in a textual file rather than a binary file.
- After the training is finished, the user can interactively explore the similarity of the words.

Run `demo-word.sh`

- Compilation process might complain about a variable "`ch`" which is not used. You may remove that variable from every `c` file that contains it.
- Script `demo-word.sh` calls `word2vec` executable to train CBOW model with dimensionality of 200, the size of the context window of 8, using negative sampling algorithm with threshold of $E-4$. Result of the analysis, i.e., word vectors, are placed in the file `vectors.bin`. Options for `cbow` parameter are 1 (`cbow`) and 0 (`skip-gram`).
- Vectors contained in file `vectors.bin` are called "**word embedding**".
- It takes some ~15 min to complete the process. You might get an error if the `text8.gz` file does not `unzip`. Just double click on the file to unzip it, and then rerun the above script.
- Finally, script invokes executable `distance` on file `vectors.bin`
- We save file `vectors.bin` in a binary format to make it smaller.

End of demo-word.sh a call to ./distance

- At the end of demo-word.sh script we invoke executable ./distance vectors.bin
- After the colon (":"), enter any word, e.g. france. You will get a long list of words closest to word france by their cosine distance (entered word)

Enter word or sentence (EXIT to break): france

Word: france Position in vocabulary: 303

| Word | Cosine distance |
|-------------|-----------------|
| ----- | ----- |
| spain | 0.672182 |
| italy | 0.621328 |
| french | 0.611584 |
| netherlands | 0.585830 |
| belgium | 0.568430 |
| provence | 0.564327 |
| alsace | 0.559103 |
| germany | 0.558620 |
| commune | 0.553380 |
| paris | 0.545077 |

- Those words are semantically closest to "France". Semantic similarity is apparently close to the geographic similarity.
- Similarity is determined by the frequency of usage in articles collected in text8 file where two words appear close one to another.

demo-word.sh, continued example of ./distance

- You can run executable distance by itself. You can enter sentences, not words only.

```
$ distance
```

```
Enter word or sentence (EXIT to break): day in berkshire # in executable distance.exe
```

```
Word: day Position in vocabulary: 137
```

```
Word: in Position in vocabulary: 5
```

```
Word: berkshire Position in vocabulary: 26294
```

| Word | Cosine distance |
|-----------------|-----------------|
| ----- | ----- |
| england | 0.472332 |
| thanksgiving | 0.439126 |
| monday | 0.424840 |
| christchurch | 0.420473 |
| yorkshire | 0.418178 |
| eastbourne | 0.404624 |
| huddersfield | 0.404338 |
| staffordshire | 0.397926 |
| gloucestershire | 0.394107 |
| fife | 0.393847 |
| hogmanay | 0.392669 |

Pre-trained word and phrase vectors

- Google published pre-trained vectors trained on part of Google News dataset (about 100 billion words). The model contains 300-dimensional vectors for 3 million words and phrases. The phrases were obtained using a simple data-driven approach described in article below. The archive is available here: [GoogleNews-vectors-negative300.bin.gz](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit).
(<https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit>)

- An example output of `./distance GoogleNews-vectors-negative300.bin:`

Enter word or sentence (EXIT to break): Chinese river

Word Cosine distance

```
Yangtze_River 0.667376
Yangtze 0.644091
Qiantang_River 0.632979
Yangtze_tributary 0.623527
Xiangjiang_River 0.615482
Huangpu_River 0.604726
Hanjiang_River 0.598110
Yangtze_river 0.597621
Hongze_Lake 0.594108
Yangtse 0.593442
```

- The above example will average vectors for words 'Chinese' and 'river' and will return the closest neighbors to the resulting vector. More examples that demonstrate results of vector addition are presented in reference below. Note that more precise and disambiguated entity vectors can be found in the dataset that uses Freebase naming.
- Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. [Distributed Representations of Words and Phrases and their Compositionality](#). In Proceedings of NIPS, 2013.*

word-analogy.exe

- We claim that the word vectors capture many linguistic regularities, which could be expressed as vector arithmetic: $\text{vector}(\text{'paris'}) - \text{vector}(\text{'france'}) + \text{vector}(\text{'italy'})$ results in a vector that is very close to $\text{vector}(\text{'rome'})$, and $\text{vector}(\text{'king'}) - \text{vector}(\text{'man'}) + \text{vector}(\text{'woman'})$ is close to $\text{vector}(\text{'queen'})$.
- We could test these assertions by running `word-analogy.exe`. We will use a bigger word vector file trained on 3-billion-word collection of text with 3 million words in vocabulary and word space of dimension 300. Once you expand the archive, type:

```
$ ./word-analogy GoogleNews-vectors-negative300.bin
```

```
Enter three words (EXIT to break): man king woman
```

```
Word: man   Position in vocabulary: 251
```

```
Word: king  Position in vocabulary: 6147
```

```
Word: woman Position in vocabulary: 641
```

| Word | Distance |
|--------------|----------|
| queen | 0.711820 |
| monarch | 0.618968 |
| princess | 0.590243 |
| crown_prince | 0.549946 |
| prince | 0.537732 |

- It appears that the age of kings and queens has not passed. The following will also work:

```
$ ./word-analogy vectors.bin
```

```
but will give different answer
```

```
Enter three words (EXIT to break): man king woman
```

```
Word: man   Position in vocabulary: 243
```

```
Word: king  Position in vocabulary: 187
```

```
Word: woman Position in vocabulary: 1013
```

| Word | Distance |
|-----------|----------|
| queen | 0.561930 |
| isabella | 0.506787 |
| marries | 0.502298 |
| betrothed | 0.498585 |
| melisende | 0.486796 |

word-analogy.exe

- Let us try one more analogy:

Enter three words (EXIT to break): **france paris russia**

Word: france Position in vocabulary: 225534

Word: paris Position in vocabulary: 198365

Word: russia Position in vocabulary: 294451

| Word | Distance |
|-----------------|-----------------|
| north_korea | 0.471760 |
| tom_cruise | 0.449580 |
| lohan | 0.448753 |
| joel | 0.445634 |
| lindsay_lohan | 0.445479 |
| heidi | 0.440514 |
| megan_fox | 0.438607 |
| britney | 0.429737 |
| russians | 0.429315 |
| moscow | 0.428812 |
| natalie_portman | 0.426762 |

- Moscow is found, indeed. Sometimes this test gives "moscow" much behind more important notions and subjects like Tom Cruise, Lindsay Lohan and Megan Fox. This vocabulary was built reading Google News and this results clearly tell you that reality shows and movies are much more important than the classical geopolitics.
- You can even say that reality shows are shaping geopolitics.
- To observe strong regularities in the word vector space, it is needed to train the models on large data set, with sufficient vector dimensionality. Using the **word2vec** tool, it is possible to train models on huge data sets (up to hundreds of billions of words). This is not sufficient evidence to link Lindsay Lohan to V. Putin.
- By the way this is an old dataset (2016).

demo-phrases.exe

- In certain applications, it is useful to have vector representations of larger pieces of text. For example, it is desirable to have only one vector for representing 'san francisco'. This can be achieved by pre-processing the training data set to form phrases using the `word2phrase` tool, as is shown in the example script `./demo-phrases.sh`.
- The `make` apparently creates a new word vector dictionary which includes phrases

```
# make
if [ ! -e news.2012.en.shuffled ]; then
    wget http://www.statmt.org/wmt14/training-monolingual-news-crawl/news.2012.en.shuffled.gz
    gzip -d news.2012.en.shuffled.gz -f
fi
sed -e "s/'/'/g" -e "s/'/'/g" -e "s/'/'/ /g" < news.2012.en.shuffled | \
    tr -c "A-Za-z'_ \n" " " > news.2012.en.shuffled-norm0
time ./word2phrase -train news.2012.en.shuffled-norm0 -output \
    news.2012.en.shuffled-norm0-phrase0 -threshold 200 -debug 2
time ./word2phrase -train news.2012.en.shuffled-norm0-phrase0 -output \
    news.2012.en.shuffled-norm0-phrase1 -threshold 100 -debug 2
tr A-Z a-z < news.2012.en.shuffled-norm0-phrase1 > news.2012.en.shuffled-norm1-phrase1
time ./word2vec -train news.2012.en.shuffled-norm1-phrase1 -output \
    vectors-phrase.bin -cbow 1 -size 200 -window 10 -negative 25 -hs 0 \
    -sample 1e-5 -threads 20 -binary 1 -iter 15
./distance vectors-phrase.bin
```

demo-phrases.sh, takes a long time

```
$ ./demo-phrases.sh
gcc word-analogy.c -o word-analogy -lm -pthread -O3 -march=native -Wall -funroll-loops -Wno-unused-result
(base) root@DESKTOP-U3NH4I2:/mnt/d/code2/word2vec# ./demo-phrases.sh
make: Nothing to be done for 'all'.
--2021-10-29 16:34:05-- http://www.statmt.org/wmt14/training-monolingual-news-crawl/news.2012.en.shuffled.gz
Resolving www.statmt.org (www.statmt.org)... 129.215.197.184
Connecting to www.statmt.org (www.statmt.org)|129.215.197.184|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 786717767 (750M) [application/x-gzip]
Saving to: 'news.2012.en.shuffled.gz'
news.2012.en.shuffl 100%[=====>] 750.27M 267KB/s in 40m 44s
2021-10-29 17:14:50 (314 KB/s) - 'news.2012.en.shuffled.gz' saved [786717767/786717767]
Starting training using file news.2012.en.shuffled-norm0
Words processed: 296M Vocab size: 33133K
Vocab size (unigrams + bigrams): 18838711
Words in train file: 296901342
Words written: 296M
real 9m20.665s
user 4m59.997s
sys 0m23.061s
Starting training using file news.2012.en.shuffled-norm0-phrase0
Words processed: 280M Vocab size: 38715K
Vocab size (unigrams + bigrams): 21728781
Words in train file: 280513979
Words written: 280M
real 11m30.978s
user 6m3.050s
sys 0m31.400s
Starting training using file news.2012.en.shuffled-norm1-phrase1
Vocab size: 681320
Words in train file: 283545447
Alpha: 0.002334 Progress: 95.33% Words/thread/sec: 107.27k
real 192m5.866s
user 625m48.253s
sys 5m1.924s
```

demo-phrases.sh, continuation

HTTP request sent, awaiting response... 200 OK

sys 5m1.924s

Enter word or sentence (EXIT to break): bay_area

Word: bay_area Position in vocabulary: 14660

| Word | Cosine distance |
|------------------------|-----------------|
| ----- | |
| san_francisco | 0.742572 |
| southern_california | 0.703172 |
| san_francisco_bay_area | 0.686204 |
| los_angeles | 0.682059 |
| northern_california | 0.674195 |
| san_diego | 0.630009 |
| california | 0.615256 |
| l_a | 0.610845 |
| orange_county | 0.597739 |

- One has an impression that this process indexes (embeds) not only individual words but bigrams as well.
- It took 4 h on my Intel i7-6700K CPU 4.00 GHz, 4 core, 8 logical units and GTX1080 Nvidia card. Output is file `vector-phrases.bin`

Once finished, Test

Alpha: 0.000005 Progress: 100.00% Words/thread/sec: 113.89k

real 84m30.377s

user 625m16.890s

sys 0m46.625s

Enter word or sentence (EXIT to break): san_francisco

Word: san_francisco Position in vocabulary: 1908

| Word | Cosine distance |
|---------------------|-----------------|
| ----- | |
| bay_area | 0.768280 |
| seattle | 0.739046 |
| los_angeles | 0.737351 |
| san_francisco's | 0.719936 |
| san_diego | 0.716911 |
| san_jose | 0.652228 |
| northern_california | 0.651532 |
| redwood_city | 0.645263 |
| new_york | 0.642389 |
| oakland | 0.639925 |

Stop words are not thrown out

- File `vector-phrases.bin` is 10X larger than file `vectors.bin` (540MB vs 56 MB). If you enter any "stop word", you get something:

```
$ ./distance vectors-phrase.bin
```

```
Enter word or sentence (EXIT to break): the
```

```
Word: the Position in vocabulary: 1
```

| Word | Cosine distance |
|--------|-----------------|
| in | 0.745736 |
| of | 0.735179 |
| which | 0.725551 |
| from | 0.629848 |
| within | 0.626430 |
| by | 0.611502 |
| a | 0.606961 |
| on | 0.603177 |
| over | 0.598575 |
| its | 0.589629 |
| for | 0.584841 |
| also | 0.572342 |
| and | 0.562596 |
| has | 0.555843 |
| as | 0.555592 |
| to | 0.551339 |
| only | 0.543157 |
| under | 0.531530 |

./distance.exe vector-phrases.bin

- The example output with the closest tokens to 'san_francisco' should look like:

```
$ ./distance.exe vector-phrases.bin
```

Word Cosine distance

```
los_angeles 0.666175  
golden_gate 0.571522  
oakland 0.557521  
california 0.554623  
san_diego 0.534939  
pasadena 0.519115  
seattle 0.512098  
taiko 0.507570  
houston 0.499762  
chicago_illinois 0.491598
```

- The linearity of the vector operations seems to weakly hold also for the addition of several vectors, so it is possible to add several word or phrase vectors to form representation of short sentences

Word Clustering

- The word vectors can be also used for deriving word classes from huge data sets. This is achieved by performing K-means clustering on top of the word vectors.
- The script that demonstrates this is `./demo-classes.sh`.
- The output is a vocabulary file with words and their corresponding class IDs, such as:

```
carnivores 234
carnivorous 234
cetaceans 234
cormorant 234
coyotes 234
crocodile 234
crocodiles 234
crustaceans 234
cultivated 234
danios 234
. . .
acceptance 412
argue 412
argues 412
arguing 412
argument 412
arguments 412
belief 412
believe 412
challenge 412
claim 412
```

Performance of the training process

- The training speed can be significantly improved by using parallel training on multiple-CPU machine (use the switch '-threads N').
- The hyper-parameter choice is crucial for performance (both speed and accuracy), however varies for different applications. The main choices to make are:
 - **architecture:** skip-gram (slower, better for infrequent words) vs CBOW (fast)
 - **the training algorithm:** hierarchical softmax (better for infrequent words) vs negative sampling (better for frequent words, better with low dimensional vectors)
 - **sub-sampling of frequent words:** can improve both accuracy and speed for large data sets (useful values are in range $1e-3$ to $1e-5$)
 - **dimensionality of the word vectors:** usually more is better, but not always
 - **context (window) size:** for skip-gram usually around 10, for CBOW around 5

How to measure quality of the word vectors

- Several factors influence the quality of the word vectors: amount and quality of the training data, size of the vectors and the training algorithm
- The quality of the vectors is crucial for any application. However, exploration of different hyper-parameter settings for complex tasks might be too time demanding.
- Mikolov et al. designed simple test sets that can be used to quickly evaluate the word vector quality.
- For the word relation test set described in [1], see `./demo-word-accuracy.sh`, for the phrase relation test set described in [2], see `./demo-phrase-accuracy.sh`.
- The accuracy depends heavily on the amount of the training data; the best results for both test sets are above 70% accuracy with coverage close to 100%.
- [1] *Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. [Efficient Estimation of Word Representations in Vector Space](#). In Proceedings of Workshop at ICLR, 2013.*
- [2] *Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. [Distributed Representations of Words and Phrases and their Compositionality](#). In Proceedings of NIPS, 2013.*

Training, Embedding, Data

The quality of the word vectors increases significantly with amount of the training data.

For research purposes, one can consider using data sets that are available on-line:

- [First billion characters from wikipedia](#) (use the pre-processing Perl script from the bottom of [Matt Mahoney's page](#))
- [Latest Wikipedia dump](#) Use the same script as above to obtain clean text. Should be more than 5 billion words. 20 GB of data.
- [WMT11 site](#): text data for several languages (duplicate sentences should be removed before training the models)
- [Dataset from "One Billion Word Language Modeling Benchmark"](#) Almost 1B words, already pre-processed text.
- [UMBC webbase corpus](#) Around 3 billion words, more info [here](#). Needs further processing (mainly tokenization).
- Text data from more languages can be obtained at [statmt.org](#) and in the [Polyglot project](#).

Pre-trained entity vectors with Freebase naming

- Google is also offering more than 1.4M pre-trained entity vectors with naming from [Freebase](#).
- This is especially helpful for projects related to knowledge mining.
- Entity vectors trained on 100B words from various news articles: [freebase-vectors-skipgram1000.bin.gz](#)
- Entity vectors trained on 100B words from various news articles, using the deprecated /en/ naming (more easily readable); the vectors are sorted by frequency: [freebase-vectors-skipgram1000-en.bin.gz](#)
- An example output of `./distance freebase-vectors-skipgram1000-en.bin:`

Enter word or sentence (EXIT to break): /en/geoffrey_hinton

Word Cosine distance

```
/en/marvin_minsky 0.457204
/en/paul_corkum 0.443342
/en/william_richard_peltier 0.432396
/en/brenda_milner 0.430886
/en/john_charles_polanyi 0.419538
/en/leslie_valiant 0.416399
/en/hava_siegelmann 0.411895
/en/hans_moravec 0.406726
/en/david_rumelhart 0.405275
/en/godel_prize 0.405176 ````
```

gensim Package

- Genism is another popular Python package for word2vec. If you have Anaconda3.5 or higher installed, it is already there. Otherwise, you would do:
- Gensime could use `vectors.bin` or similar file with trained word vectors, so just copy it to the directory where you want to run a `gensim` script.

- To run a king - man + woman like query, write a script with this code:

```
# import modules & set up logging
import gensim, logging
# define logging level. We only care about warnings
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
level=logging.WARNING)
# Import already existing vectors from previous example
model = gensim.models.KeyedVectors.load_word2vec_format('vectors.bin',
binary=True)
# Get word similarity using out of the box task
most_similar = model.most_similar(positive=['emperor', 'woman'],
negative=['man'])
print(most_similar)

[('empress', 0.6104894876480103), ('theodora', 0.5391228199005127),
('emperors', 0.5160053968429565), ('chlorus', 0.5084700584411621),
('saimei', 0.4915323853492737), ('montferrat', 0.47812795639038086),
('suiko', 0.4621468484401703), ('comnena', 0.4601095914840698),
('faustina', 0.4584055542945862), ('constantius', 0.45779654383659363)]
```

If you do not trust the Technology

Female names returned by the previous query are all empresses or princesses (daughters of emperors and empresses):

- Saimei ((655–661) and Ashikaga Okiko were Japanese Empresses.
- Faustina was the Empress and wife of Roman Emperor Antoninus Pius.
- Anna Comnena (Komnena) was a Byzantine princess and a daughter of an Emperor and an Empress.

You can find all of this in Wikipedia. One could conclude that Google's techies include Wikipedia into their corpuses 😊

GloVe: Global Vectors for Word Representation

- Pennington et al. argue that the online scanning approach used by word2vec is suboptimal since it doesn't fully exploit statistical information regarding word co-occurrences. They demonstrate a *Global Vectors* (GloVe) model which combines the benefits of the word2vec skip-gram model when it comes to word analogy tasks, with the benefits of matrix factorization methods that can exploit global statistical information.
- The GloVe model...
- *... produces a vector space with meaningful substructure, as evidenced by its performance of 75% on a recent word analogy task. It also outperforms related models on similarity tasks and named entity recognition.*
- The source code for the model, as well as trained word vectors can be found at <http://nlp.stanford.edu/projects/glove/>

Spark NLP

Spark NLP

- The following is practically an advertisement for a very powerful MLP framework, Spark NLP by John Snow Labs.
- Spark NLP uses distributed high-performance computing power and scalability of Apache Spark to solve most common NLP problems.
- Spark NLP puts to full use Spark ML Pipelines for data processing and analysis.
- NLP pipeline is always just a part of a bigger data processing pipeline: For example, question answering involves
 - loading training data,
 - transforming data,
 - applying NLP annotators,
 - building features,
 - training the value extraction models,
 - evaluating the results (train/test split or cross-validation), and
 - hyperparameter estimation.
- None of competing NLP Toolkits (APIs) does all of this in such comprehensive manner.

What is Spark NLP

- Spark NLP is an open-source natural language processing library, built on top of Apache Spark and Spark ML.
- Spark NLP provides an easy API to integrate with ML Pipelines and it is commercially supported by **John Snow Labs**.
- Spark NLP's annotators utilize rule-based algorithms, machine learning with Tensorflow running under the hood to power specific deep learning implementations.
- The library covers many common NLP tasks, including tokenization, stemming, lemmatization, part of speech tagging, sentiment analysis, spell checking, named entity recognition, and more.
- The full list of annotators, pipelines, and concepts is described in the online reference. All of them are included as open-source and can be used by training models with your data.
- Spark NLP also provides pre-trained pipelines and models, although they serve as a way of getting a feeling on how the library works, and not for production use.
- Spark NLP library is written in Scala and it includes Scala and Python APIs for use from Spark. It has no dependency on any other NLP or ML library.

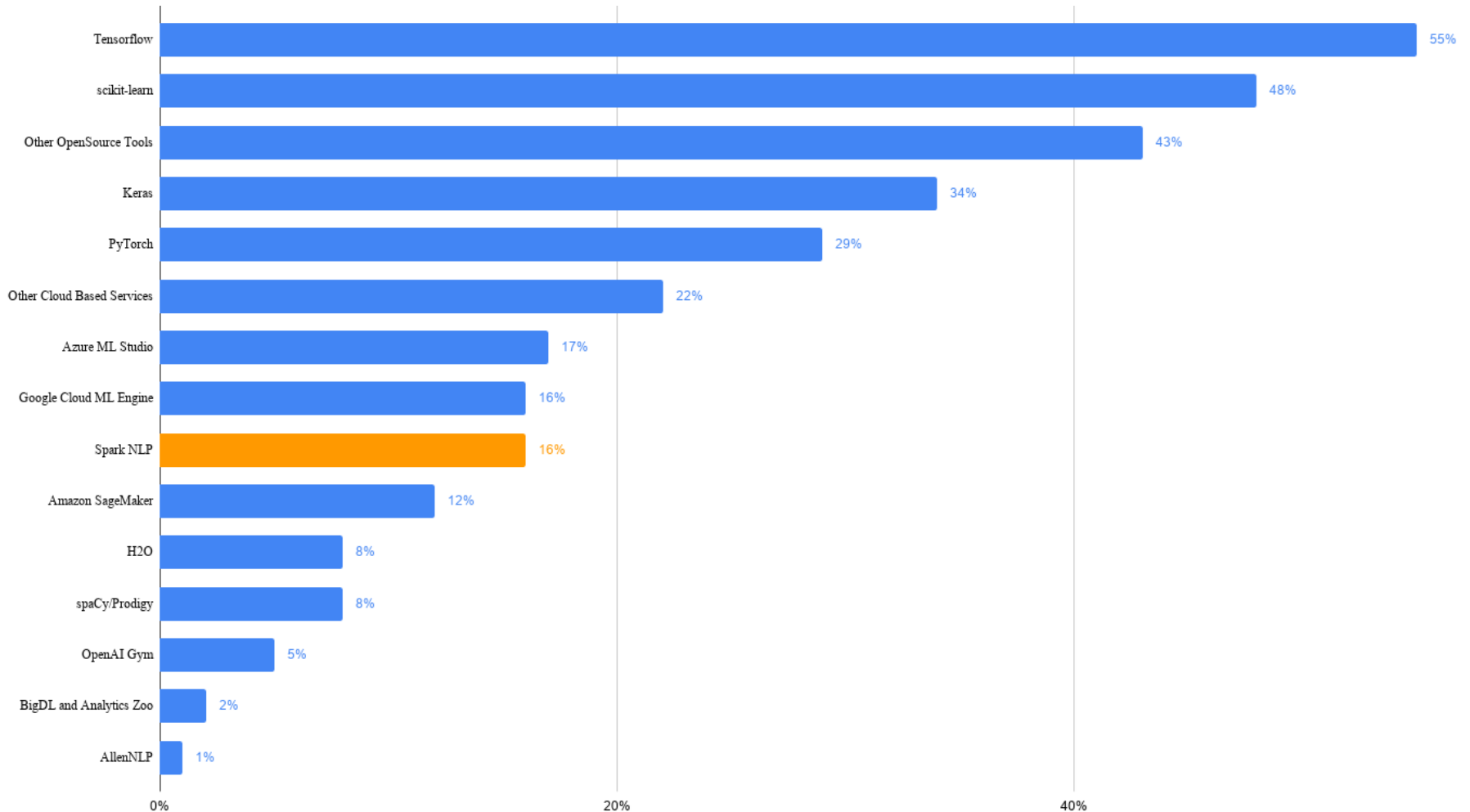
Feature Comparison

- Of all APIs, Spark NLP offers the largest number of standard NLP features:

| Name | Spark NLP | spaCy | NLTK | CoreNLP |
|--------------------|-----------|-------|------|---------|
| Sentence detection | Yes | Yes | Yes | Yes |
| Tokenization | Yes | Yes | Yes | Yes |
| Stemming | Yes | Yes | Yes | Yes |
| Lemmatization | Yes | Yes | Yes | Yes |
| POS tagger | Yes | Yes | Yes | Yes |
| NER | Yes | Yes | Yes | Yes |
| Dependency parse | Yes | Yes | Yes | Yes |
| Text matcher | Yes | Yes | No | Yes |
| Date matcher | Yes | No | No | Yes |
| Chunking | Yes | Yes | Yes | Yes |
| Spell checker | Yes | No | No | No |
| Sentiment detector | Yes | No | No | Yes |
| Pretrained models | Yes | Yes | Yes | Yes |
| Training models | Yes | Yes | Yes | Yes |

Popularity

- Spark NLP appears to be the most popular NLP library in production use.



Transfer Learning

- With Spark NLP we can take advantage of transfer learning and implementing the latest and greatest algorithms and models in NLP research.
- Transfer learning is a means to extract knowledge from a source setting and apply it to a different target setting.
- Transfer learning is an effective way to keep improving the accuracy of NLP models and to get reliable accuracies even with small data by leveraging the already existing labelled data of some related task or domain. As a result, there is no need to collect millions of data points in order to train a state-of-the-art model.
- Spark NLP includes direct access to pre-trained models most modern NLP related Deep Learning architectures (models) such as: [ELMo](#), [BERT](#), [RoBERTa](#), [ALBERT](#), [XLNet](#), [Ernie](#), [ULMFiT](#), [OpenAI transformer](#), which are all open-source. These can be tuned or reused without a major computing effort.

System Architecture

- Apache Spark is the big-data platform of choice for enterprises because of its ability to process large streaming data. Spark provides real-time stream processing, interactive processing, graph processing, in-memory processing as well as batch processing with very fast speed, ease of use and standard interface.
- Apache Spark has a module called [Spark ML](#) which introduces several ML components, most notably **Estimators**, which are trainable algorithms and **Transformers** which are either a result of training an estimator, or an algorithm that doesn't require training at all.
- Both Estimators and Transformers can be part of a **Pipeline**, which is a sequence of steps that execute in order and are probably depending on each other's result.
- **Spark-NLP** introduces **NLP annotators** that merge within this framework and its algorithms are meant to predict in parallel.
- All **Annotators** are either Estimators or Transformers as we see in Spark ML. An Estimator in Spark ML is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.
- A Transformer is an algorithm which can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer that transforms a DataFrame with features into a DataFrame with predictions

System Architecture of Spark NLP

- As described on previous slide Spark NLP is built a top of Apache Spark ML and its pipelines, TensorFlow Graphs and various Deep Learning Models for text processing.



References

- [1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. [Efficient Estimation of Word Representations in Vector Space](#). In Proceedings of Workshop at ICLR, 2013.
- [2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. [Distributed Representations of Words and Phrases and their Compositionality](#). In Proceedings of NIPS, 2013.
- [3] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. [Linguistic Regularities in Continuous Space Word Representations](#). In Proceedings of NAACL HLT, 2013.
- Tomas Mikolov, Quoc V. Le and Ilya Sutskever. [Exploiting Similarities among Languages for Machine Translation](#). We show how the word vectors can be applied to machine translation. Code for improved version from Georgiana Dinu [here](#).
- Word2vec in Python by Radim Rehurek in [gensim](#) (plus [tutorial](#) and [demo](#) that uses the above model trained on Google News).
- Word2vec in Java as part of the [deeplearning4j](#) project. Another Java version from Medallia [here](#).
- Word2vec implementation in [Spark MLlib](#).
- Comparison with traditional count-based vectors and cbow model trained on a different corpus by [CIMEC UNITN](#).