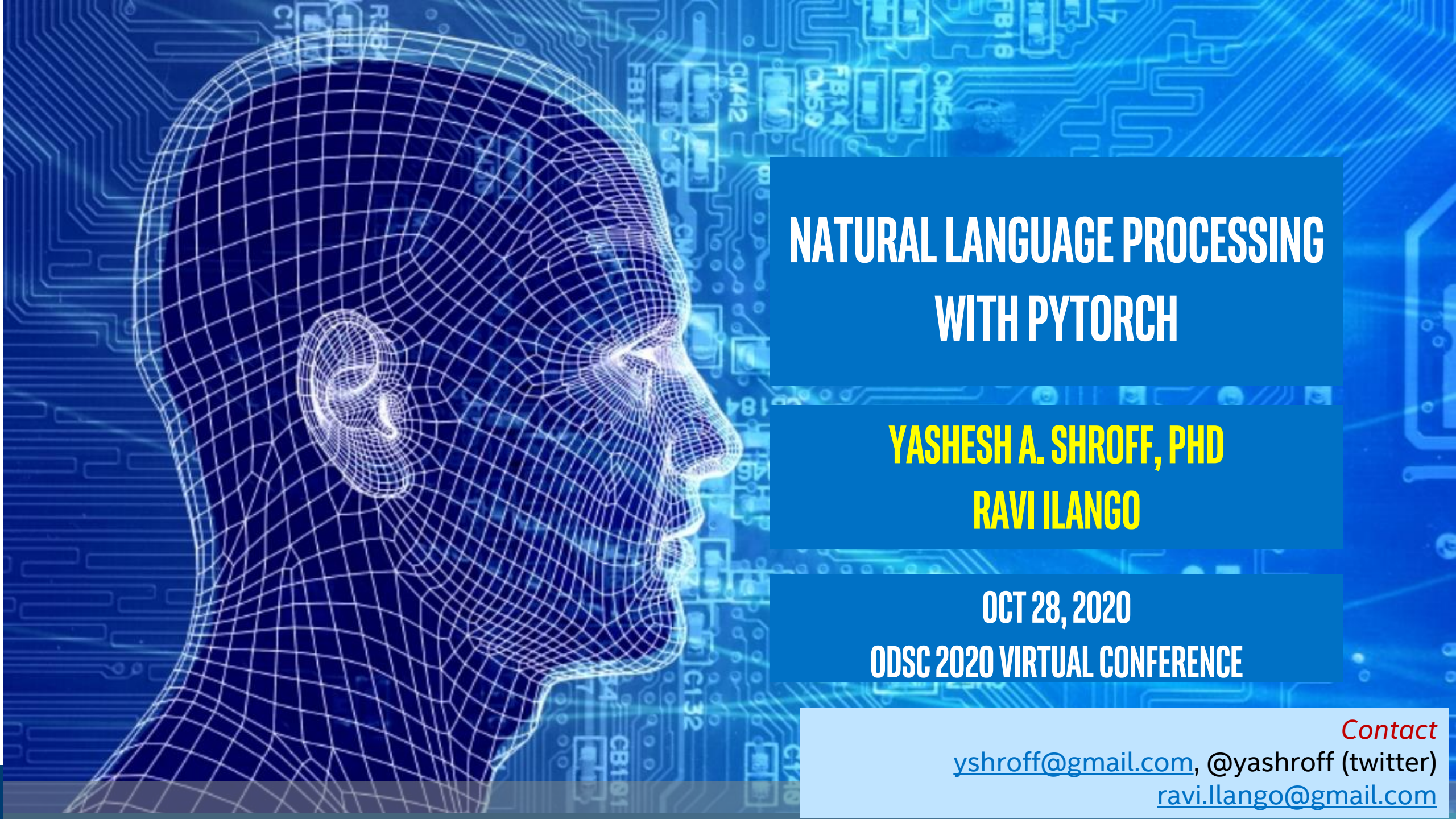


NATURAL LANGUAGE PROCESSING WITH PYTORCH

YASHESH A. SHROFF, PHD
RAVI ILANGO

OCT 28, 2020
ODSC 2020 VIRTUAL CONFERENCE

Contact
yshroff@gmail.com, @yashroff (twitter)
ravi.Ilango@gmail.com

Training Overview

Part I - Fundamentals & Applications of Natural Language Models

- Natural Language Understanding Overview
  - A brief *primer* on technology, compute, and constraints
- Classification in NLP
  - Lab
- Probabilistic Models
  - Lab
- Sequence Models
  - Lab

Training Overview

- Introduction to SpaCy
  - Popular library for tokenization
  - Lab

Transformers – Part 2

- Application with Text Classifier
  - Using document vectorization & Sklearn classifier
- Improving with Word Embeddings & LSTM
  - Lab + DL based approach to Word Embedding (training on the fly)
- Introduction to pre-trained models, ex. BERT (Huggingface)
  - Lab
- Text Summarization
  - Lab
- AirFlow
  - Applying data pipelines in production

Labs
1. NLP Intro
2. SpaCy (POS, Tokenization)
3. Text Classification (Python, Sklearn, TF-IDF)
4. Text Classification (LSTM, Tokenizer)
5. Text Classification (XLNet)
6. Text Summarization (NLTK, Gensim)

A word about the training (setting expectations for the next 3 hours)

What we cover:

- Deep Learning based Neural Machine Translation approach with some theoretical background and heavy labs usage
- Covers modern (last 2-4 years) development in NLP
- Gives a practitioner's perspective on how to build your NLP pipeline

What we do not cover much beyond foundational context:

- Statistical and probabilistic approach (minimal)
- Early Neural Machine Translation approaches (marginal)

“You shall know a word by the company it keeps”

J.R. Firth, 1957

Context is important if you want to understand the meaning of a word

Yashesh A. Shroff

Bit about me:

- Working at Intel as a Strategic Planner, responsible for driving ecosystem growth for AI, media, and graphics on discrete GPU platforms for the Data Center
- Prior roles in IOT, Mobile Client, and Intel manufacturing
- Academic background:
  - ~15 published papers, 5 patents
  - PhD from UC Berkeley
  - MBA from Columbia Graduate School of Business
  - Intensely passionate about programming & product development
- Contact:
  - @yashroff
  - yshroff@gmail.com

Growing use of NLP

Speech <-> Text

- Translation of text into spoken words and vice-versa
- Apps: Siri, Alexa, Google Home

Machine Translation

- Translating from one language to another
- Apps: Google Translate

Text summarization

- Concise version of long text, support
- Apps: Chatbots, AI legal services, Reddit bots

Search

- Understanding what the user wants
- Context aware

Setting up your Environment

Most of the lab work will be in the Python Jupyter notebooks in workshop Github repo:

- Jupyter (https://jupyter.org/install)
- PyTorch (https://pytorch.org/get-started/locally/#start-loca
- SpaCy (https://spacy.io/usage)
- Hugging face transformer (https://huggingface.co/transformers/installation.html)

Training GitHub Repo: https://github.com/ravi-ilango/odsc2020_nlp

Install git on your laptop: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
And run the following command: `git clone https://github.com/ravi-ilango/odsc2020_nlp`
Use conda or pipenv to install the requirements dependencies in a virtual environment.

```
import numpy as np
import matplotlib.pyplot as plt

conda create -n pynlp python=3.6
source activate pynlp
conda install ipython
conda install -c conda-forge jupyterlab
conda install pytorch torchvision -c pytorch
pip install transformers
```

A brief history of Machine Translation

Pre-2012: Statistical Machine Translation

- Language modeling, Probabilistic approach
- Con: Requires “high-resource” languages

Neural Machine Translation

- word2vec
- GloVe
- ELMo
- Transformer

Underlying common approaches

- Model, Training data, Training process

NMT: Key Papers

- word2vec: Mikolov et. al. (Google)
- GloVe: Pennington et al., Stanford CS. EMNLP 2014
- ElMo:
- ELMo (Embeddings from Language Models)
  - Memory augmented deep learning
- Survey paper (https://arxiv.org/abs/1708.02709)
  - Blog (https://medium.com/dair-ai/deep-learning-for-nlp-an-overview-of-recent-trends-d0d8f40a776d)
- Vaswani et al., Google Brain. December 2017.
  - *The Illustrated Transformer* blog post
  - *The Annotated Transformer* blog post

Ref: https://eigenfoo.xyz/transformers-in-nlp/

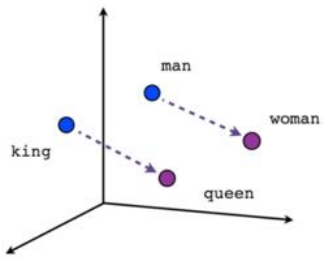
Classical vs. DL NLP

Classical:

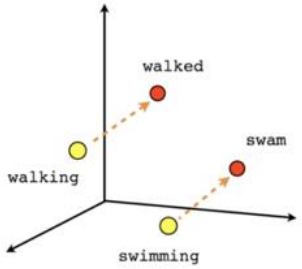
- Task customization for NLP Applications

DL Based NLP

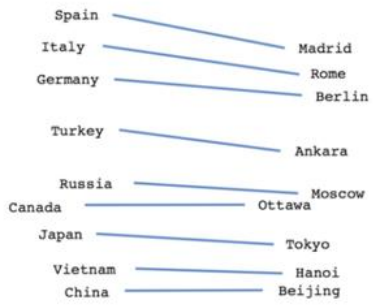
- Compressed representation
- Word Embeddings



Male-Female

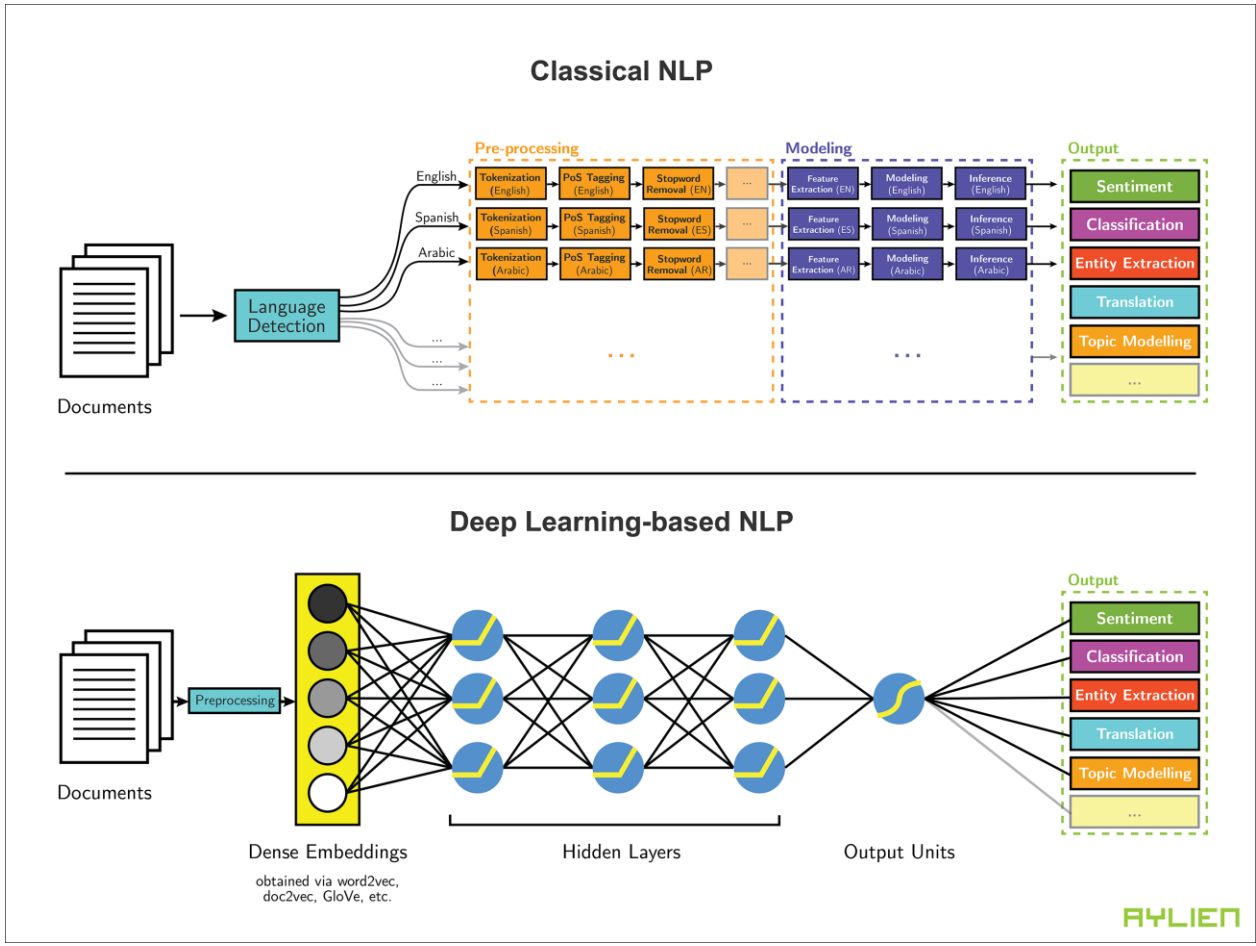


Verb tense



Country-Capital

Reference: https://arxiv.org/abs/1301.3781
(Efficient Estimation of Word Representations in Vector Space)



Reference: https://aylien.com/blog/leveraging-deep-learning-for-multilingual

Getting the Text Ready (corpus)

Start with clean text, without immaterial items, such as HTML tags from web scraped corpus.

Normalize

- Normalize text by converting it to all lower case, removing punctuation, & extra white spaces

Tokenize

- Split text into words, n-grams, or phrases (tokens)

"I love morning runs"
- Unigrams: "I", "love", "morning", "runs"
- Bigrams (n=2): "I love", "love morning", "morning runs"
- Trigrams (n=3): "I love morning", "love morning runs"

Remove Stop words

- Remove common words like "a", "the", "and", "on", etc.

Stemming

- Convert to stem

ex. Dancer, dancing, dance become 'danc'
Studies, Study, Studying: Stud

POS, NER

- Identify Parts of Speech (POS), such as verb, noun, named entity
- Lemmatization: root word (am, are, is >> be)

Example: Raw tweet	Preprocessed output
@huggingface is building a fantastic library of NLP datasets and models at http://huggingface.com	Build fantastic library NLP dataset model

Starting from scratch

Normalization: convert every letter to a common case so each word is represented by a unique token

```
text = text.lower()
text = re.sub(r"[^a-zA-Z0-9]", " ", text)
```

Token: Implies symbol, splitting each sentence into words

```
text = text.split()
```

```
from nltk.tokenize import
word_tokenize
words = word_tokenize(text)
```

NLTK: Split text into sentences

```
from nltk.tokenize import sent_tokenize
sentences = sent_tokenize(text)
```

Stop-word removal

Stop-word removal

```
from nltk.corpus import stopwords
print(stopwords.words("english")
words = [w for w in words if not in stopwords.words("english")
```

Parts of speech tagging

```
from nltk import pos_tag
sentence = word_tokenize("Start practicing with small code.")
pos_text = pos_tag(sentence)
```

Name Entity Recognition (NER) to label names (used for indexing and searching for news articles)

```
from nltk import ne_chunk
ne_chunk(pos_text)
```

Normalizing word variations

1. Stemming: reducing words to their stem or root

```
from nltk.stem.porter import PorterStemmer
stemmed = [PorterStemmer().stem(w) for w in words]
print(stopwords.words("english")
words = [w for w in words if not in stopwords.words("english")
```

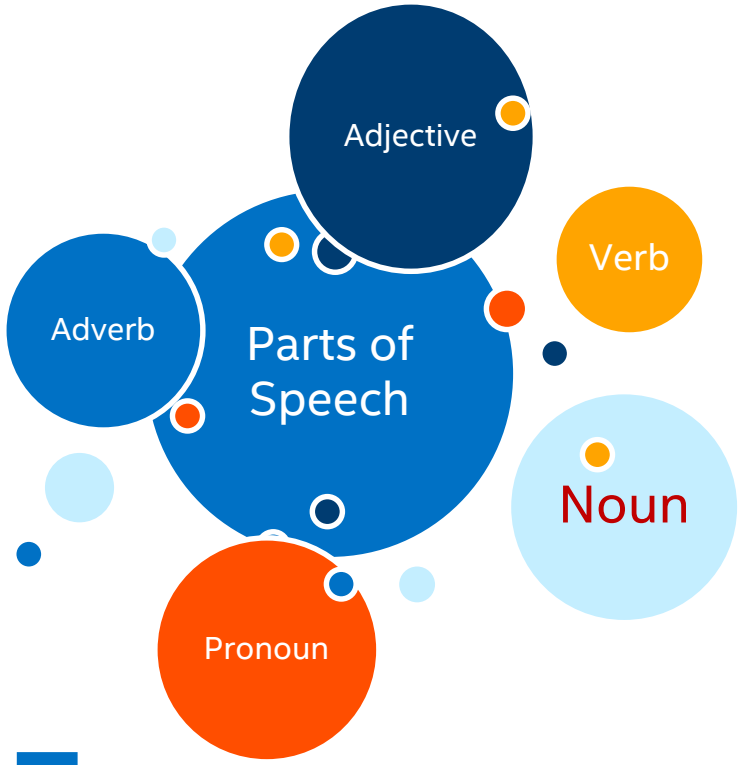
2. Lemmization

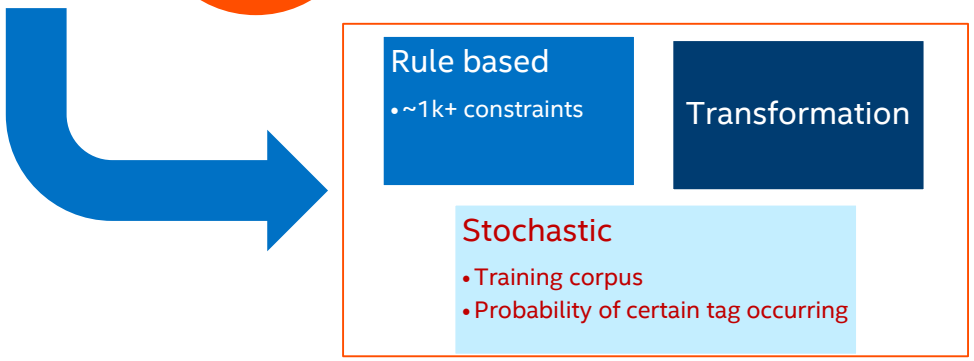
```
from nltk.stem.wordnet import WordNetLemmatizer
lemmed = [WordNetLemmatizer().lemmatize(w) for w in words]
lemmed = [WordNetLemmatizer().lemmatize(w, pos='v') for w in lemmed]
```

Name Entity Recognition (NER) to label names (used for indexing and searching for news articles)

```
from nltk import ne_chunk
ne_chunk(pos_text)
```

Parts of Speech Tagging





One tag for each part of speech

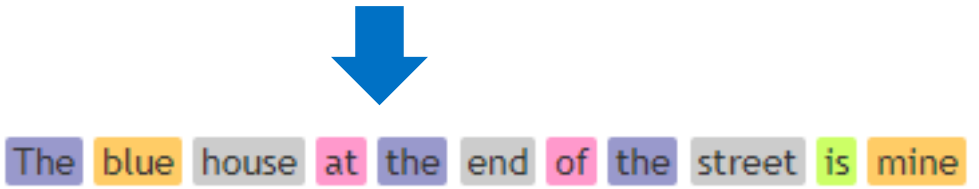
- Choose a courser tagset (~6 is useful)
- Finely grained tagsets exist (ex. Upenn Tree Bank II)

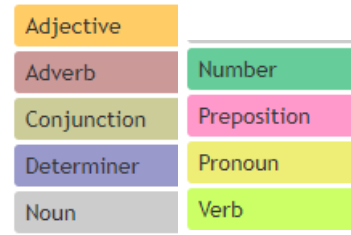
Sentence: "Flies like a flower"

- flies: Noun or Verb?
- like: preposition, adverb, conjunction, noun or verb?
- a: article, noun, or preposition
- flower: noun or verb?

https://parts-of-speech.info/

"The blue house at the end of the street is mine."





Word Embeddings

Techniques to convert text data to vectors

Frequency based

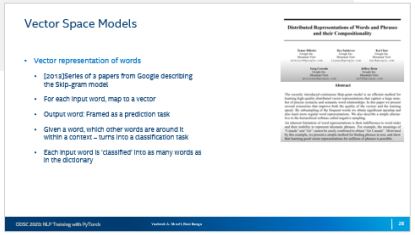
- Count Vector
- TF-IDF
- Co-occurrence Vector

- Count based feature engineering strategies (bag of words models)
- Effective for extracting features
- Not structured
  - Misses semantics, structure, sequence & nearby word context
- 3 main methods covered in this lecture. There are more...

Prediction based Word2Vec

- CBOW
- Skip-Gram

- Capture meaning of the word
- Semantic relationship with other adjacent words
  - Deep Learning based model computes distributed & dense vector representation of words
- Lower dimensionality than bag of words model approach
- **Alternative:** GloVe



Word Embedding

Frequency based

Count Vector

Doc 1	"The athletes were playing"
Doc 2	"Ronaldo was playing well"

	The	Athlete	was	playing	Ronaldo	well
Doc 1	1	1	1	1	0	0
Doc 2	0	0	1	1	1	1

- Real-world corpus can be millions of documents & 100s M unique words resulting in a very sparse matrix.
- Pick top 10k words as an alternative.

TF-IDF vectorization

Document 1: "This is about cars"
Document 2: "This is about kids"

Term	Count		TF-IDF
	Doc1	Doc2	Doc 1 example
This	2	1	2/8*log(2/2) = 0
is	3	2	3/8*log(2/2) = 0
about	1	2	1/8*log(2/2) = 0
Kids	0	4	
cars	2	0	2/8*log(2/1) = 0.075
Terms	8	9	

$$TF = \frac{\#\ times\ term\ T\ appears\ in\ the\ document}{\#\ of\ terms\ in\ the\ document, m}$$

$$IDF = \left(\frac{Number\ of\ documents, N}{Numer\ of\ documents\ in\ which\ term\ T\ appears, n}\right) = \log\left(\frac{N}{n}\right)$$

Calculate $TF\ x\ IDF$

- Term frequency across corpus accounted, but penalizes common words
- Words appearing only in a subset of document are weighed favorably

Co-Occurrence Vector

"He is not lazy. He is intelligent. He is smart"

	He	is	not	lazy	intelligent	smart
He	0	4	2	1	2	1
is	4	0	1	2	2	1
not	2	1	0	1	0	0
lazy	1	2	1	0	0	0
intelligent	2	2	0	0	0	0
smart	1	1	0	0	0	0

$$\underset{m\ x\ n}{\hat{X}} \approx \underbrace{\underset{m\ x\ r}{U}\ \underset{r\ x\ r}{S}}_{\text{Word-vector representation}}\ \underbrace{\underset{r\ x\ n}{V^T}}_{\text{Context}}$$

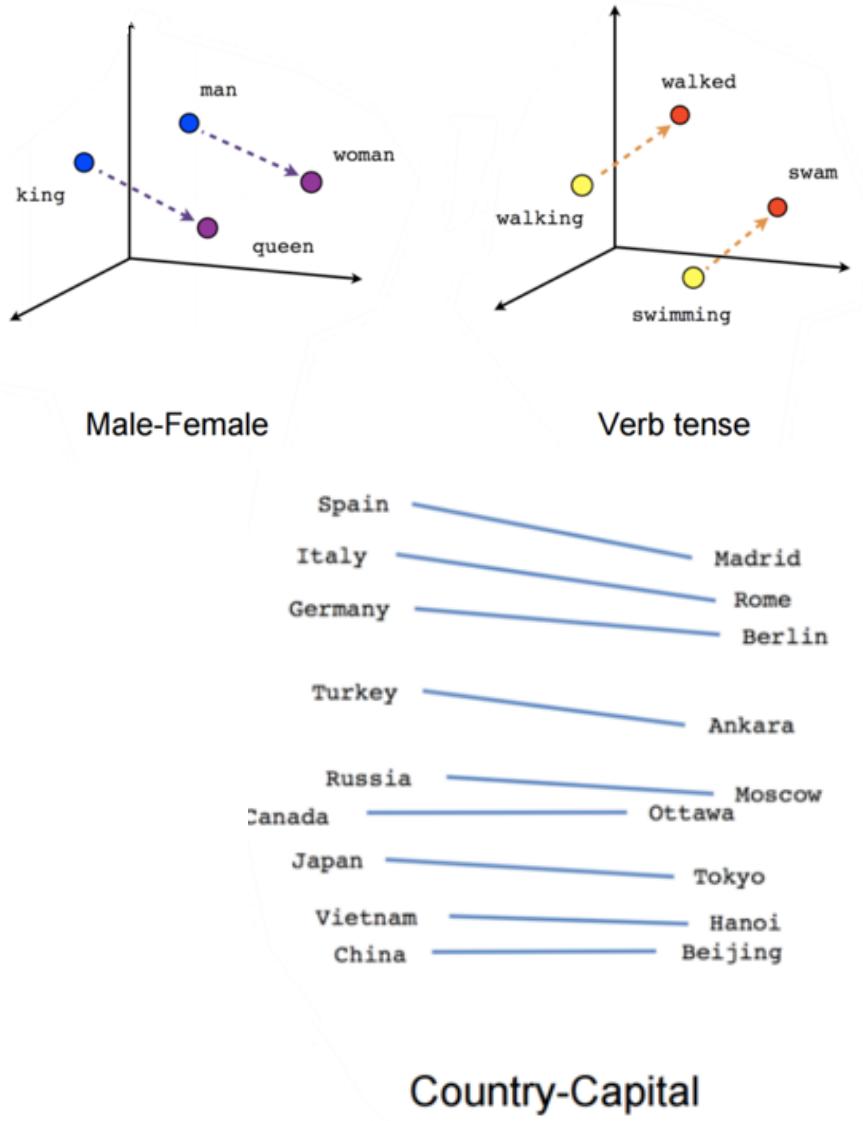
m: # of terms
n : m minus stop words
- Uses SVD decomposition and PCA to reduce dimensionality

- Similar words tend to occur together: "Airbus is a plane", "Boeing is a plane"
- Calculates the # of times words appear together in a context window

Prediction based Word Embedding

Key Idea: Words share context

- Embedding of a word in the corpus (numeric representation) is a function of its related words – words that share the same context
- Examples: “word” => (embeddings)
  - “car” => (“road”, “traffic”, “accident”)
  - “language” => (“words”, “vocabulary”, “meaning”)
  - “San Francisco” => (“New York”, “London”, “Paris”)

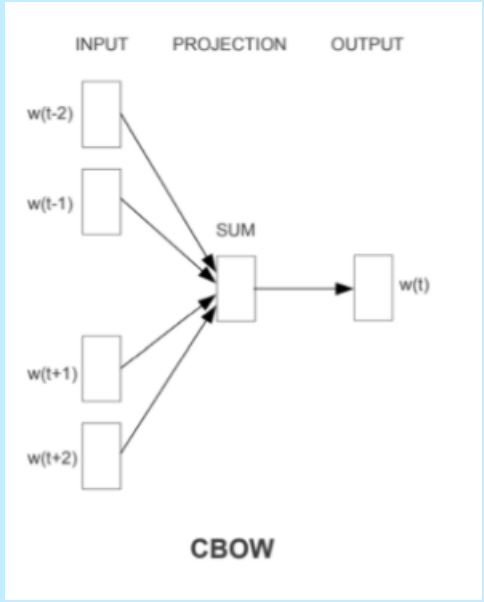


Reference: https://arxiv.org/abs/1301.3781
(Efficient Estimation of Word Representations in Vector Space)

Word Embedding

Prediction based Word2Vec

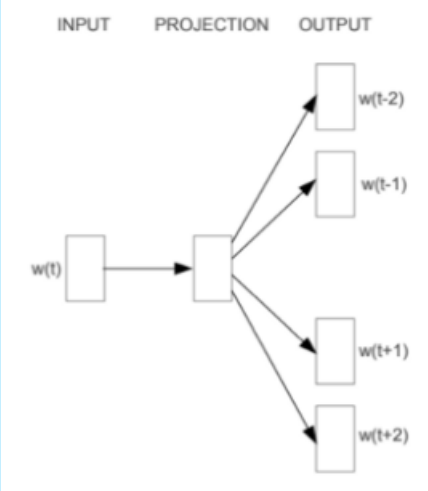
CBOW



https://arxiv.org/pdf/1301.3781.pdf

- The distributed representation of the surrounding words are combined to predict the word in the middle
- Input word is OHE vector of size V and hidden layer is of size N
- Pairs of context window & target window
- Using context window of 2, let's parse:
  - "The quick brown fox jumps over the lazy dog"
    - "quick __ fox": ([quick, fox], brown)
    - "the __ brown": ([the, brown], quick)
- Tip: Use a framework to implement (ex. Gensim)

Skip-Gram



- The distributed representation of the input word is used to predict the context
- Mikolov (Google) introduced in 2013
- Works well with small data but CBOW is faster
- Using context window of 2, let's parse:
  - "The quick brown fox jumps over the lazy dog"
    - "__ brown __" (brown => [quick, fox])
    - "___ quick ___" (quick => [the, brown])

Top NLP Packages

NLTK

- Preprocessing: Tokenizing, POS-tagging, Lemmatizing, Stemming
- Cons: Slow, not optimized

Gensim

- Specialized, optimized library for topic-modeling and document similarity

SpaCy

- ”Industry-ready” NLP modules.
- Optimized algorithms for tokenization, POS tagging
- Text parsing, similarity calculation with word vectors

Huggingface – Transformers / Datasets (will cover later)

Python, SpaCy → Go to the website & follow the tutorial (commands)

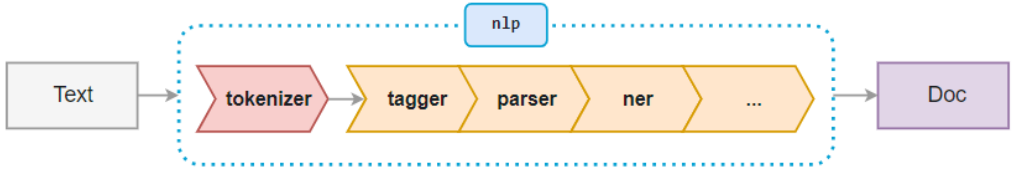
Objective:

- Covered in lecture
  - Word–Embedding. Tokenization:
- NER: showing country
- POS
- Powered Regex with NER

SpaCy Lab

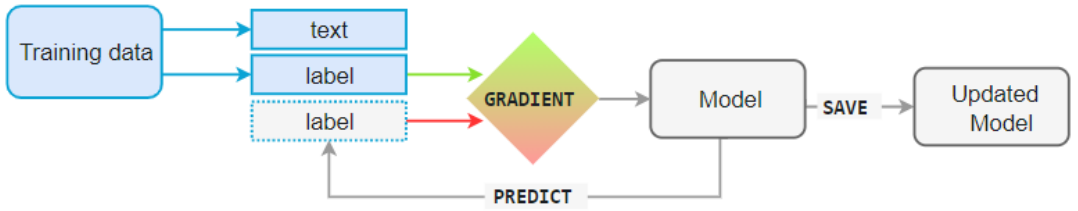
Language Processing Pipelines

- SpaCy's `nlp` class first tokenizes the text
- Default pipeline: tagger, parser, NER
- Can add custom components at any point in the pipeline
- Finally, produce a `Doc` object



Training Models

- SpaCy's `nlp` class first tokenizes the text
- Default pipeline: tagger, parser, NER
- Can add custom components at any point in the pipeline
- Finally, produce a `Doc` object



Is there a way to automate the flow?

Reference: spacy.io

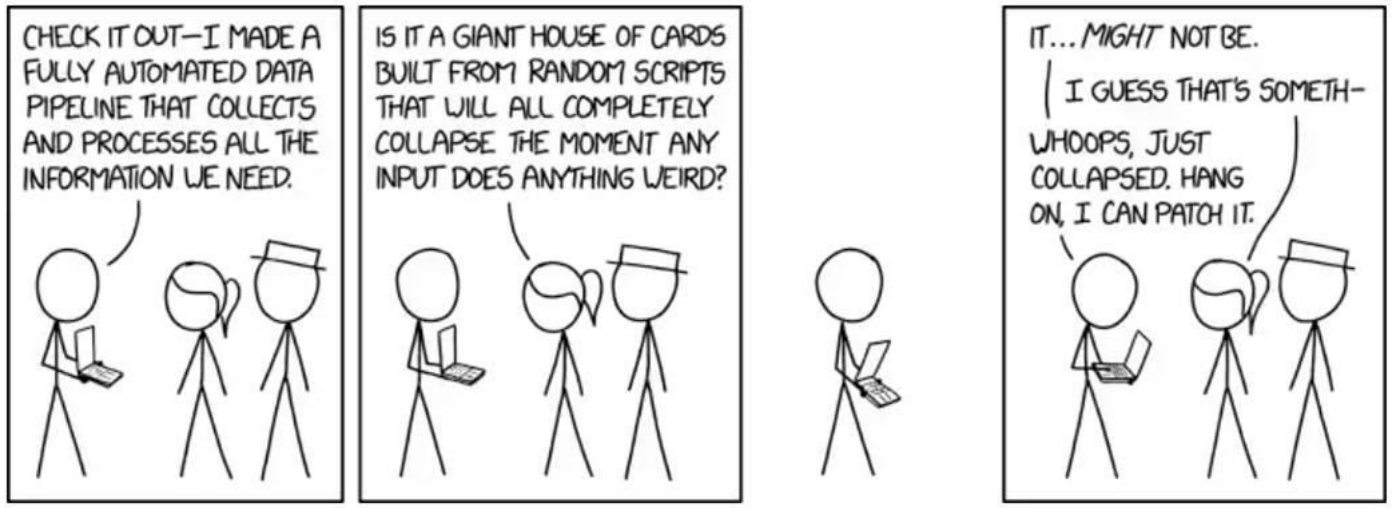
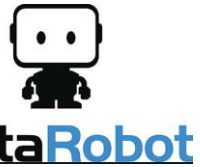


Image source: [xkcd: Data Pipeline](https://xkcd.com/2054/)

Creating NLP pipelines

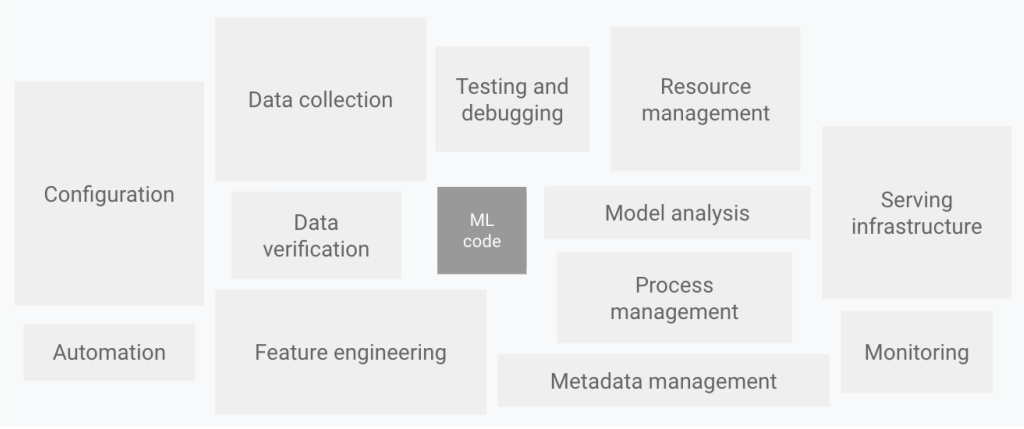


Problem statement:

- Building a deep learning model is a small part of an end-to-end cycle of deploying an app
- Building an NLP pipeline is critical in managing model versions, dataset versions, and ensuring resiliency of the infrastructure

Directed Acyclic Graph, or DAG, to the rescue

- DAG is a data pipeline, an ETL process, or a workflow
- Each node or task of DAG includes an operator: Python, Bash, etc.
- When to use:
  - Going beyond cron jobs
  - Usually when business logic demands it



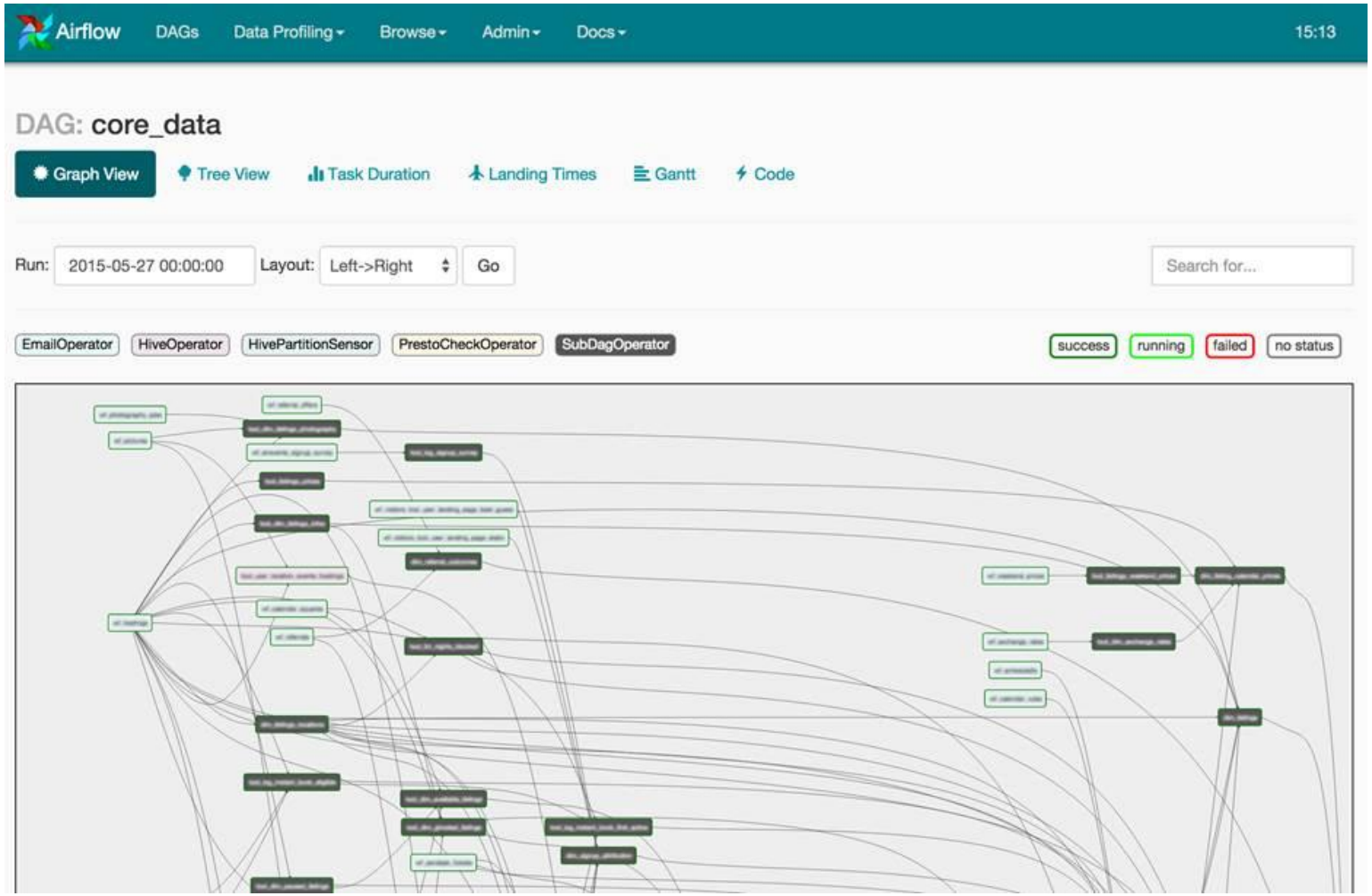
Source: https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

Airflow: Getting started

```
# Install apache-airflow
sudo pip3 install apache-airflow
export AIRFLOW_HOME="/Users/<HOMEDIR>/airflow_learning/airflow"
airflow scheduler
# in a different terminal, run:
airflow webserver
```

How it looks in practice

- Data warehousing: Organize & clean input text
- A/B testing (trying out different models)
- Business Policy & governance compliance

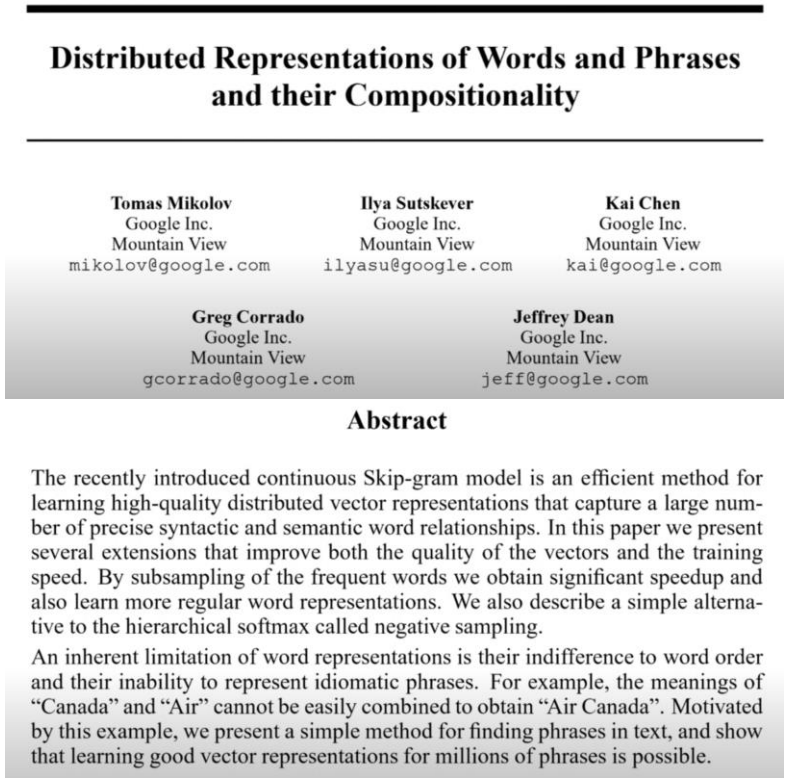


Goto:
https://airflow.apache.org/docs/stable/tutorial.html

Backup

Vector Space Models

- Vector representation of words
  - [2013]Series of 3 papers from Google describing the Skip-gram model
  - For each input word, map to a vector
  - Output word: Framed as a prediction task
  - Given a word, which other words are around it within a context – turns into a classification task
  - Each input word is ‘classified’ into as many words as in the dictionary



Distributed Representations of Words and Phrases and their Compositionality

Tomas Mikolov
Google Inc.
Mountain View
mikolov@google.com

Ilya Sutskever
Google Inc.
Mountain View
ilyasu@google.com

Kai Chen
Google Inc.
Mountain View
kai@google.com

Greg Corrado
Google Inc.
Mountain View
gcorrado@google.com

Jeffrey Dean
Google Inc.
Mountain View
jeff@google.com

Abstract

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. In this paper we present several extensions that improve both the quality of the vectors and the training speed. By subsampling of the frequent words we obtain significant speedup and also learn more regular word representations. We also describe a simple alternative to the hierarchical softmax called negative sampling.

An inherent limitation of word representations is their indifference to word order and their inability to represent idiomatic phrases. For example, the meanings of “Canada” and “Air” cannot be easily combined to obtain “Air Canada”. Motivated by this example, we present a simple method for finding phrases in text, and show that learning good vector representations for millions of phrases is possible.

References

Models

- FastText: https://fasttext.cc/
- CBOW and Skip-gram comparison: https://fasttext.cc/docs/en/unsupervised-tutorial.html
- NLTK vs SpaCy: https://www.activestate.com/blog/natural-language-processing-nltk-vs-spacy/
- Geek4Geeks Python Word Embeddings: https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/
- Multi-lingual text analysis: https://aylien.com/blog/leveraging-deep-learning-for-multilingual
- Hero NLP libraries: https://elitedatascience.com/python-nlp-libraries
- Huggingface Universal Sentence Embeddings: https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a

Word Embeddings

- Language modeling technique used for mapping words to vectors of real numbers. It represents words or phrases in vector space with several dimensions. Word embeddings can be generated using various methods like neural networks, co-occurrence matrix, probabilistic models, etc.

Word2Vec

- Consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer and one output layer. Word2Vec utilizes two architectures :
  - **CBOW (Continuous Bag of Words)** model predicts the current word given context words within specific window. The input layer contains the context words, and the output layer contains the current word. The hidden layer contains the number of dimensions in which we want to represent current word present at the output layer.
  - **Skip Gram** predicts the surrounding context words within specific window given current word. The input layer contains the current word, and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word present at the input layer.

Ref: https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/

Ref: https://www.quora.com/Is-word2vec-basically-an-autoencoder

Backup

Difference between Autoencoders & word2vec

	Autoencoder	Word2Vec
Dense Representation	An auto-encoder's job is to represent a (sparse) input dataset in a compressed form that retains the most relevant information such that it may be reconstructed at the output with minimal loss from the compressed representation. In order to do this, the input data is subjected to an information bottleneck so that the encoder is able to learn the most efficient 'latent representation' of the input rather than just memorizing the input. This is similar in spirit to matrix factorization.	Converts a sparse unique indexing of the vocabulary (i.e. in the input text, each word in the vocabulary is represented as unique index in a dictionary) to a continuous, dense, and distributed representation that can be considered to be a compressed 'latent representation' of words in the vocabulary. This can also be considered to be similar in spirit to matrix factorization, but the goal is to encode the context around the word rather than the word itself.
Type of Network	Originally proposed, the encoder and decoders are simple, fully connected, feed forward neural nets, but nothing prevents replacement of the networks by CNNs, RNNs and other deep net architectures	Originally proposed for simple, shallow, fully connected feed forward networks since the goal was to allow for fast training from large amounts of data, but similar to auto encoders, nothing prevents replacement of the network by deep net, RNNs or other architectures.
Learning Problem	Is a self-supervised learning problem because there are no explicit labels. The input dataset also serves as the output label since the goal is reconstruction.	is an unsupervised problem (corpus of unlabelled text) posed as a binary classification problem because the goal is predict the source context words given the target word (skip-gram) or predict the target word given the source context (CBOW).
Input Dataset	Can be applied to any sort of input dataset where learning a dense representation is useful.	Specifically used only for words in natural language, but nothing prevents using the method from using the technique for other sparse representations where learning the context is important/useful.
Loss Function	learn by back-propagating the reconstruction loss from decoder to encoder.	learn by back-propagating the gradient from the soft-max classifier to the dense word vectors such that the cross entropy loss of the classifier is minimized

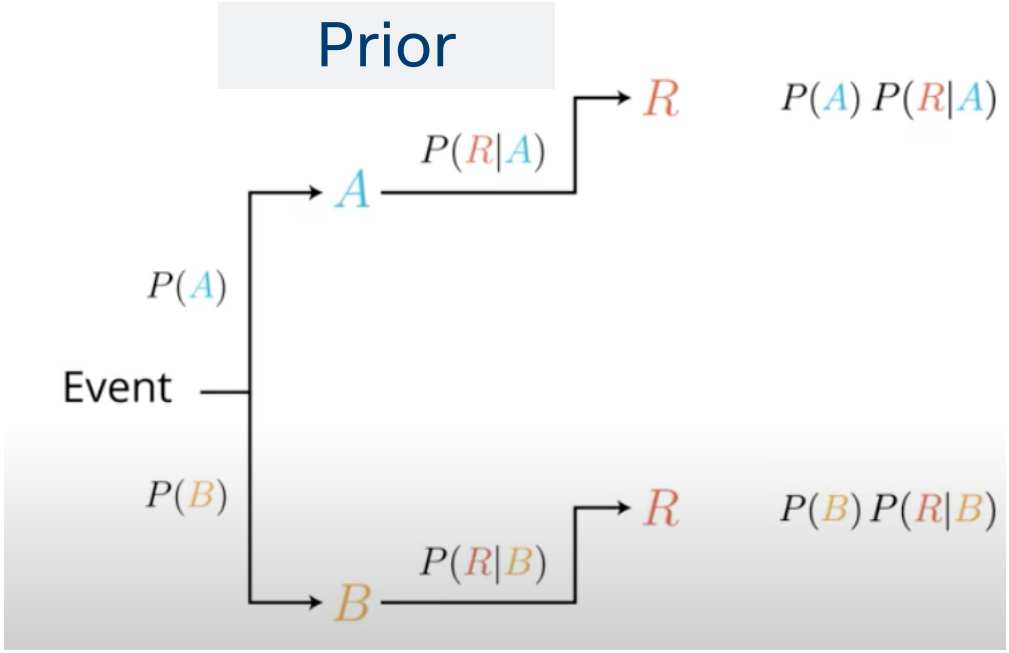
Word2Vec Code Implementation

https://fasttext.cc/docs/en/unsupervised-tutorial.html

- CBOW and Skip Gram implementations
- Get the data:
  - wget https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2

- Trains 1GB of Wiki data; not feasible for our purpose
- See Ravi's training for Word2Vec overview

Bayes Theorem



Since these two probabilities do not add to one, we just divide them both by their sum so that the new probabilities now do add to one.

1. Naïve assumption: assume that our probabilities are independent.
   - P(A&B) = P(A)*P(B)
2. Conditional probability – the basis for our theorem
   - P(A|B)*P(B) = P(B|A)*P(A) => P(A|B) α P(B|A)*P(A)