Processing "Text" for Machine Learning

(NLP: Natural Language Processing)

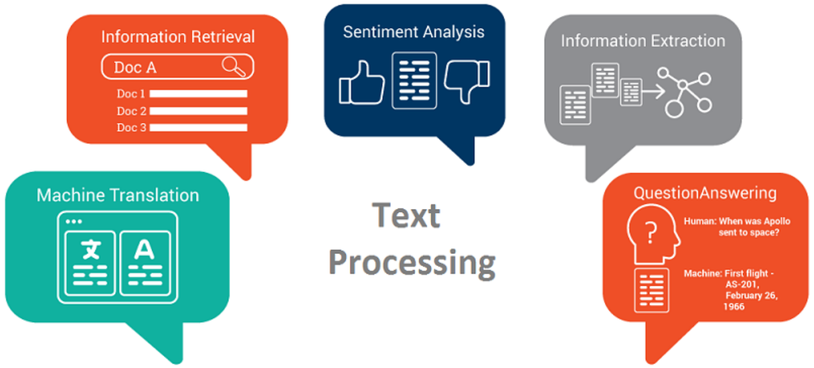
Dr. Ratna Babu Chinnam

Industrial & Systems Engineering

Wayne State University

Primary Source: Javaid Nabi | [Link]

What is Text Processing?



- A common task in many ML applications.
- Task of transforming text into something an algorithm can digest can be complicated.
- Example Applications:
    - **Text Classification & Ranking**: Goal is to predict a class (label) of a document (e.g., spam filtering: e-mail is spam or not), or rank documents based on their relevance.
    - **Sentiment Analysis**: Aims to determine the attitude or emotional reaction of a person with respect to some topic -- e.g., positive or negative attitude, anger, sarcasm. It is broadly used in customer satisfaction studies (e.g., analyzing product reviews).
    - **Document Summarization**: Is a set of methods for creating short, meaningful descriptions of long texts (e.g., research papers or reports).
    - **Named Entity Extraction**: Algorithms process a stream of unstructured text and recognize predefined categories of objects (entities) in it, such as a person, company name, date, price, title etc. It enables faster text analysis by transforming unstructured information into a structured, table-like (or JSON) form.
    - **Natural Language Understanding**: Used for transforming a human-generated text into more formal representations interpretable by a computer.
    - **Machine Translation**: Task of automatically translating text or speech from one human language into another.

Source: sigmoidal | [Link]

Typical Steps

- **Text Preprocessing:**
  - An important step that transforms text into a more digestible form so that machine learning algorithms can perform better
- **Feature Extraction**:
  - Words of the text represent discrete, “categorical” features
  - Involves mapping/encoding of textual data to “real valued” vectors for use by algorithms
- **Machine Learning**:
  - Address the ultimate task/application

Text Preprocessing: Typical Steps

- **Tokenization**: Converts sentences to lists of words
- **Removing unnecessary punctuation**: E.g., html tags
- **Removing "stop words":** Frequent words such as ”the”, ”is”, etc. that do not have specific semantic value
- **Stemming**: Words are reduced to a "root" by removing "inflection" through dropping unnecessary characters, usually a suffix
  - Stemmed form of "studies" is: "studi"
  - Stemmed form of "studying" is: "study"
- **Lemmatization:** Another approach to remove inflection by determining part of speech and utilizing detailed language database
  - Lemmatized form of "studies" is: "study"
  - Lemmatized form of "studying" is: "study"

Data Preprocessing: Python

- NLTK: Popular NLP library useful for all sorts of tasks from tokenization, stemming, tagging, parsing, and beyond
- BeautifulSoup: Library for extracting data from HTML and XML documents

```
import nltk
from nltk.tokenize import word_tokenize

#Function to split text into words
tokens = word_tokenize("The quick brown fox jumps over the lazy dog")
print(tokens)
```

OUT: [‘The’, ‘quick’, ‘brown’, ‘fox’, ‘jumps’, ‘over’, ‘the’, ‘lazy’, ‘dog’]

```
from nltk.corpus import stopwords
stop_words = set(stopwords.words(‘english’))
tokens = [w for w in tokens if not w in stop_words]
print(tokens)
```

OUT: [‘The’, ‘quick’, ‘brown’, ‘fox’, ‘jumps’, ‘lazy’, ‘dog’]

```
#NLTK provides several stemmer interfaces
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer(); stems = []
for t in tokens:
    stems.append(porter.stem(t))
print(stems)
```

OUT: [‘the’, ‘quick’, ‘brown’, ‘fox’, ‘jump’, ‘lazi’, ‘dog’]

Extracting Features from Text: *Bag of Words (BOW)*

- **Vocabulary**: List of unique words in text corpus
- Represent each sentence or document as a vector with each word represented as 1/0 for **present/absent** from vocabulary
  - What is the dimensionality of the ML model's input vector?
- Another representation can **count** number of times each word appears in a document
- Most popular approach is using the **Term Frequency–Inverse Document Frequency (TF-IDF)** technique
  - **Term Frequency (TF)** = (# of times term t appears in a document) / (# of terms in the document)
  - **Inverse Document Frequency (IDF) =** $\log(N/n)$, where, N is # of documents and n is # of documents a term t has appeared in
    - IDF of a rare word is high, whereas IDF of a frequent word is likely to be low
  - Calculate $\mathbf{TF{-}IDF}_{\mathrm{Term}}$ = **TF * IDF**

Extracting Features from Text: *TF-IDF*

- Example Text Corpus: 2 Documents

Document 1

Term	Count
This	1
is	1
a	1
beautiful	2
day	5

Document 2

Term	Count
This	1
is	1
a	1
beautiful	1
night	2

- **TF**('**beautiful**’, **Document1**) = 2/10, **IDF**('**beautiful**')=log(2/2) = 0
- TF(‘**day**’,Document1) = 5/10, IDF(‘day’)=log(2/1) = 0.30

- **TF-IDF**(‘**beautiful**’, **Document1**) = (2/10)*0 = 0
- TF-IDF(‘**day**’, Document1) = (5/10)*0.30 = 0.15

TF-IDF: Document Classification using Sickit-Learn

WAYNE STATE UNIVERSITY

Task: Classifying "[20 Newsgroup]" dataset

```
import sklearn

#Load dataset
from sklearn.datasets import
fetch_20newsgroups as 20ng
20_train = 20ng(subset='train', shuffle=True)

#List categories and inspect a record
20_train.target_names #prints categories
print("\n".join(20_train.data[0].split("\n")[:
3])) #prints first line of first data file

#Extract features
from sklearn.feature_extraction.text import
CountVectorizer
count_vect = CountVectorizer()
X_train_counts =
count_vect.fit_transform(20_train.data)
X_train_counts.shape
from sklearn.feature_extraction.text import
TfidfTransformer
tfidf_trans = TfidfTransformer()
X_train_tfidf =
trans.fit_transform(X_train_counts)
X_train_tfidf.shape
```

```
#Build Model (Linear SVM) & Pipeline
from sklearn.linear_model import
SGDClassifier*
from sklearn.pipeline import Pipeline

text_clf_svm = Pipeline([('vect',
CountVectorizer()),('tfidf',
TfidfTransformer()),('clf',
SGDClassifier(loss='hinge',penalty='l2',alpha=
1e-3,n_iter=5,random_state=42)),])

#Fit model
text_clf_svm.fit(20_train.data,
20_train.target)
#Make predictions for test data
predicted_svm =
text_clf_svm.predict(20_test.data)
#Calculate Test Accuracy
np.mean(predicted_svm == 20_test.target)
```

Accuracy should be > 80% without removing stop words or stemming or hyper-parameter training!

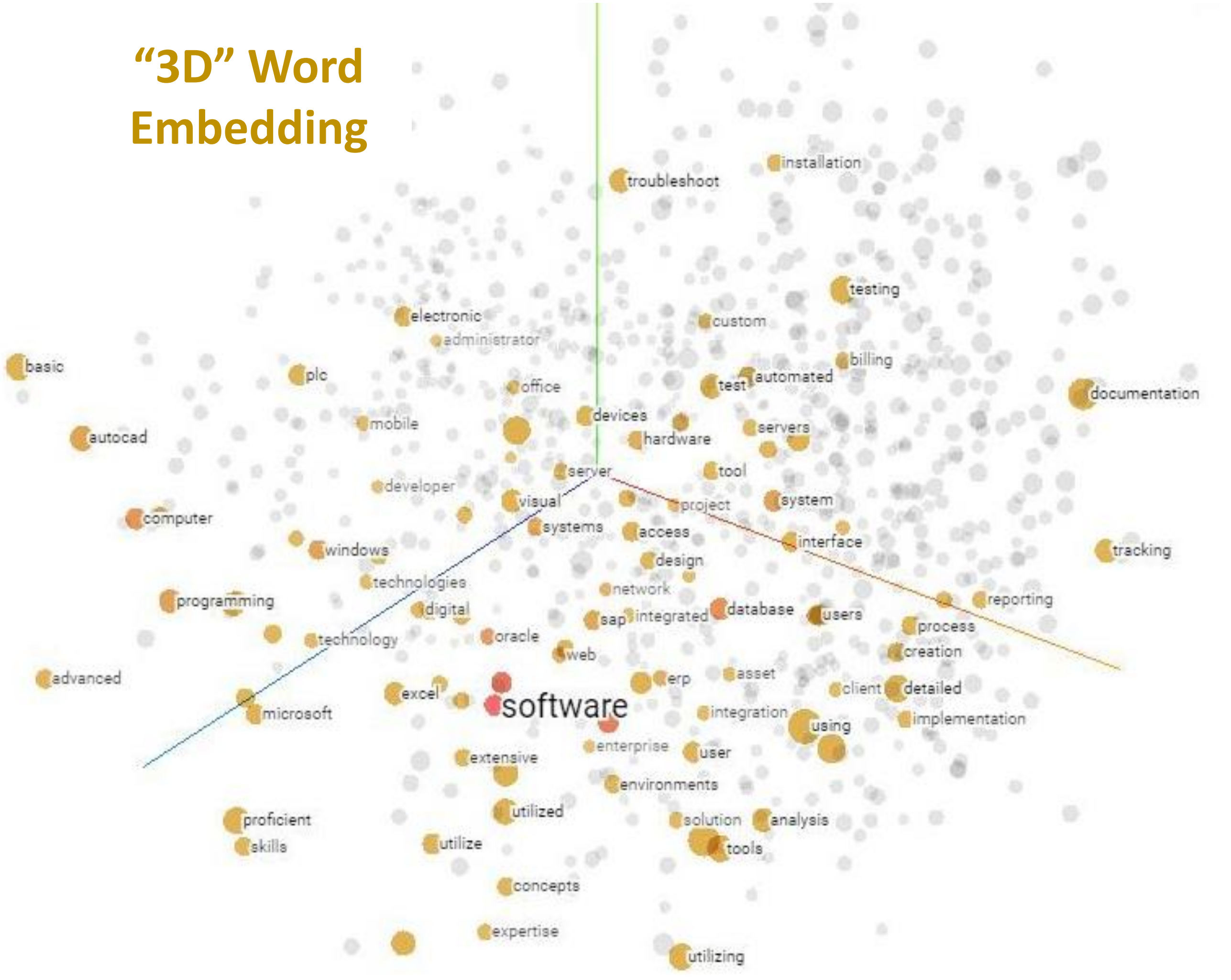
```
#To Remove Stop Words, Update Pipeline:
CountVectorizer(stop_words='english')
```

* This estimator implements regularized linear models with stochastic gradient descent (SGD) learning. Link

Extracting Features from Text: *Word Embedding*

- Disadvantage of using **BOW** is that it **discards word order**
  - Ignores "context" and word meaning
  - Curse of dimensionality
- **Word Embedding**: Numerical representation of words
  - Related words, **based on a corpus of relationships**, are placed together
- **Embeddings:**
  - Can be learnt from "scratch"
    - Need more data
    - Insert "embedding layer" into network
  - "Pre-trained" embeddings
    - Embeddings available with different dimensions

"3D" Word Embedding



Source: Aakash Chotrani

“Pre-Trained” Word and Sentence Embeddings

WAYNE STATE UNIVERSITY

Word2Vec (2013) by Google

- Uses Neural Network for word representations. Two popular variants:
  - **Continuous Bag of Words** (**CBOW**): Predict a given word given its neighboring words (context)
  - **Skip Gram**: Solve reverse problem — given target words, try to come up with neighboring words
- Desirable benefits:
  - Original model trained on 1.6B word dataset and 1M words in the vocabulary
  - Even people with modest amount of data could use these pre-calculated embeddings in their specific tasks and walk away with a lot of improvement
  - Numerical representation of words with similar meaning were close (had small cosine distance)
  - “Surprisingly, similarity of word representations goes beyond simple syntactic regularities
    - Example: *vector(”King”) — vector(”Man”) + vector(”Woman”)* results in a vector that is closest to the vector representation of the word *Queen*

GloVe (2014) by Stanford

- **Combines benefits of** both — global matrix factorization methods (like LSA), which do well on capturing **statistical structure** and **local context** windows methods like Word2Vec, which do well on analogy tasks.
- Instead of extracting numerical representations from training a neural network for certain task like predicting the next word, GloVe vectors have inherent meaning, which is **derived from word co-occurrences**.

fastText (2016) by Facebook

- Conceptually similar to Word2Vec, but with a twist — instead using words for creating embeddings, it uses n-gram of characters
  - Representation of word “test” with $n = 2$ would be $< t, te, es, st, t >$
- Approach allows it to **generalize to unknown words**, or the words which were not part of vocabulary in training
- Requires lesser training data compared to Word2Vec

Sentences and paragraphs can be embedded through word vector averaging and other techniques!

“Pre-Trained” Word Embeddings: GloVe

- Smallest package of embeddings is 822Mb, called “*glove.6B.zip*”
  - Trained on a dataset of 1B tokens (words) with a vocabulary of 0.4M words
  - Different embedding vector sizes: 50, 100, 200 and 300 dimensions
- After downloading and unzipping, you will see a few files, one of which is “*glove.6B.100d.txt”:* Contains a 100-d version of embedding
- We can seed the Keras “*Embedding”* layer with weights from the pre-trained embedding for the words in training dataset
- If you peek inside the file, you will see a token (word) followed by the weights (100 numbers) on each of 0.4M lines for the 0.4M words. Example: “the”

  the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953 -0.39141 0.3344 -0.57545 …

- Keras provides a Tokenizer class that can be fit on the training data, can convert text to sequences consistently by calling the *texts_to_sequences()* method on the *Tokenizer* class, and provides access to the dictionary mapping of words to integers in a *word_index* attribute
- Loading the embedding can be slow
  - It might be better to filter the embedding for the unique words in your dataset.
- We need to create a matrix of one embedding for each word in the training dataset
  - Need to enumerate all unique words in the Tokenizer.word_index and locating the embedding weight vector from the loaded GloVe embedding

Source: Jason Brownlee | Blog: Link

Word Embedding Layers for DL with Keras

WAYNE STATE UNIVERSITY

Task: Classifying "Synthetic" Dataset using GloVe

```
from numpy import array, asarray, zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense, Flatten, Embedding
# define documents
docs = ['Well done!','Good work','Great effort',
     'nice work','Excellent!','Weak',
     'Poor effort!','not good','poor work',
     'Could have done better.'] #Longest document (4 words)
# define class labels
labels = array([1,1,1,1,1,0,0,0,0,0])
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs) #print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length,
padding='post') #print(padded_docs)
# load the whole GloVe embedding into memory
embeddings_index = dict()
f = open('../glove_data/glove.6B/glove.6B.100d.txt')
for line in f:
   values = line.split()
   word = values[0]
   coefs = asarray(values[1:], dtype='float32')
   embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

```
# create embedding weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
   embedding_vector = embeddings_index.get(word)
   if embedding_vector is not None: #Otherwise, zeros
     embedding_matrix[i] = embedding_vector
# define sequential DL model
model = Sequential()
e = Embedding(vocab_size, 100, weights=[embedding_matrix],
input_length=4, trainable=False)
model.add(e); model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
# summarize the model
print(model.summary())
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

If learning embedding from scratch:

```
model.add(Embedding(vocab_size,8,
input_length=4))
```

OUTPUT:

```
Encoded Docs ->  [[6, 2], [3, 1], [7, 4], [8, 1], [9], [10],
                 [5, 4], [11, 3], [5, 1], [12, 13, 2, 14]]

Padded Docs ->   [[ 6  2  0  0]
                  [ 3  1  0  0]
                  [ 7  4  0  0]
                    . . .
                  [12 13  2 14]]

GloVe Vocabulary Word Vectors -> Loaded 400,000 word vectors.

Network Configuration ->  Layer (type)             Output Shape      Param #
                          ============================================
                          embedding_1 (Embedding)  (None, 4, 100)    1500
                          flatten_1 (Flatten)      (None, 400)       0
                          dense_1 (Dense)          (None, 1)         401

Parameters ->  Total params: 1,901
               Trainable params: 401
               Non-trainable params: 1,500

Accuracy ->    Accuracy: 100.0 %
```

Source: Jason Brownlee | Blog: Link

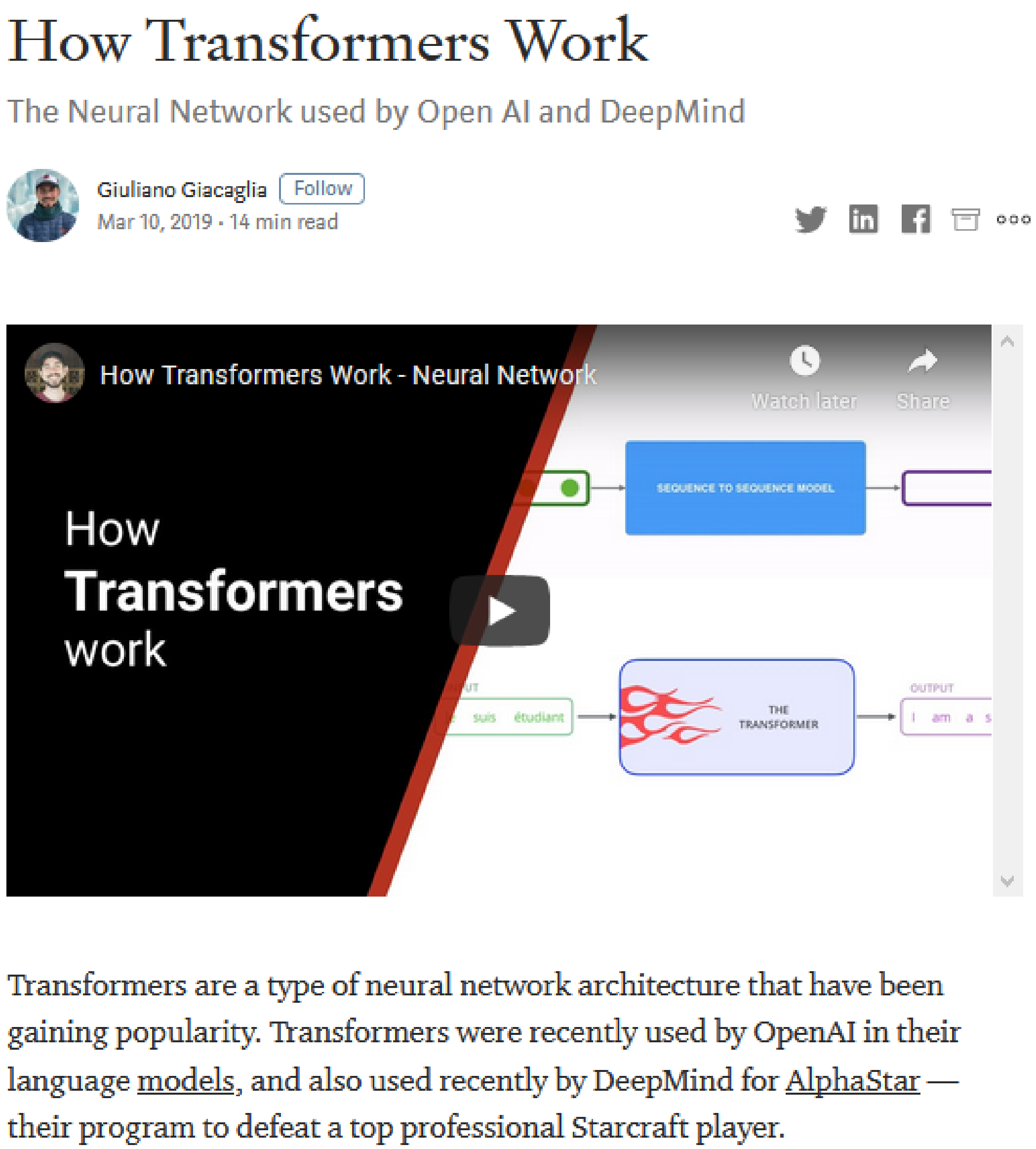
Case Study: Sentiment of IMDb Movie Reviews

- Javaid Nabi: Python, Link
  - No embedding
  - 79% accuracy
- TensorFlow Core: Python, Link
  - Uses tf.keras and learns embedding from scratch
  - 87% accuracy
- TensorFlow Hub: Python, Link
  - Uses tf.keras and pre-trained embedding obtained using TensorFlow Hub, a library and platform for transfer learning
  - 85% accuracy
- More advanced approaches should get closer to 95% accuracy!
  - Recurrent networks (including LSTM and GRU), attention and transformer frameworks!

IMDB dataset contains the text of 50,000 movie reviews from the Internet Movie Database. These are split into 25,000 reviews for training and 25,000 reviews for testing. The training and testing sets are *balanced*, meaning they contain an equal number of positive and negative reviews.

“Transformers” for NLP

- A type of neural network architecture gaining popularity for NLP
- Developed to solve the problem of sequence transduction, or neural machine translation.
  - Any task that transforms an input sequence to an output sequence (e.g., speech recognition, text-to-speech transformation, etc.)
- Excellent Tutorial by Giuliano Giacaglia: [Link]



How Transformers Work

The Neural Network used by Open AI and DeepMind

Giuliano Giacaglia Follow
Mar 10, 2019 · 14 min read

Transformers are a type of neural network architecture that have been gaining popularity. Transformers were recently used by OpenAI in their language models, and also used recently by DeepMind for AlphaStar — their program to defeat a top professional Starcraft player.