### **Lecture Notes**

### **on**

### **Vector Space Models of Semantics**

**George Pashev**

**<http://gpashev.com>**

#### 1. Introduction to Vector Space Models (VSMs)

* ****Definition and Core Concept****: Vector Space Models (VSMs) are computational frameworks that represent text data, such as words, phrases, or entire documents, as vectors in a multidimensional space. Each dimension typically corresponds to a unique term from the corpus, allowing documents and words to be represented by points or vectors within this space. The primary insight behind VSMs is that semantic similarity between text units can be inferred from the geometric relationships between their corresponding vectors, such as distances or angles.
* ****Importance in Text Processing****: The ability of VSMs to quantify semantic similarity plays a crucial role in numerous natural language processing (NLP) tasks, including search engines, document classification, and sentiment analysis. By transforming qualitative text data into quantitative vectors, VSMs bridge the gap between human language and computational algorithms, facilitating more effective text analysis and understanding.

#### 2. Motivation for VSMs

* ****Challenges in Language Understanding****: The inherent ambiguity and complexity of human language pose significant challenges for computational systems. Subtleties of context, polysemy (words with multiple meanings), and the richness of linguistic expressions make it difficult for computers to grasp the meaning of text data. VSMs offer a structured, quantitative approach to representing text that helps mitigate these challenges by focusing on patterns of word usage and their co-occurrences.
* ****Wide-ranging Applications****: Beyond enhancing search engine algorithms, VSMs underpin advancements in machine translation, content recommendation systems, and automatic summarization, among other applications. By enabling machines to process and analyze text at scale, VSMs contribute to more intelligent, context-aware computational systems.

#### 3. The Structure of VSMs

* ****Term-Document Matrix****: This matrix structure represents documents as vectors, with each element indicating the presence or frequency of a term within a document. It serves as the basis for many information retrieval systems, where the goal is to find documents most relevant to a query.
* ****Word-Context Matrix****: Here, words are represented as vectors based on their contexts — the surrounding words or linguistic constructs. This model facilitates analyses of word meanings and relationships by examining the contexts in which words appear.
* ****Pair-Pattern Matrix****: Extending the word-context model, this structure focuses on pairs of words and the patterns of their co-occurrence. It's particularly useful for understanding the relationships between words (e.g., synonyms, antonyms) and for tasks like analogy solving and relationship extraction.

#### 4. From Frequency to Meaning: Constructing VSMs

* ****Tokenization****: The process of breaking down text into its constituent tokens (e.g., words, phrases). Effective tokenization is foundational for VSM construction, as it determines the basic units of analysis for the model.
* ****Normalization****: To reduce variability in the data, tokens are often normalized through stemming (reducing words to their root form) or lemmatization (reducing words to their base or dictionary form). This step helps in grouping together different forms of the same word, ensuring that they contribute to the same vector dimensions.
* ****Annotation****: Adding metadata to tokens, such as part-of-speech tags or syntactic roles, enhances the model's ability to distinguish between different uses of the same word and to capture more nuanced semantic relationships.
* ****Weighting****: Applying schemes like TF-IDF or PMI adjusts the raw frequency counts based on the importance and distinctiveness of terms within the corpus. Weighting helps in emphasizing terms that are more informative about the semantic content of text units.

#### 5. Mathematical Processing in VSMs

* ****Building the Frequency Matrix****: The initial construction of the matrix involves counting the occurrences of each token within specific contexts or documents, forming the raw data from which semantic insights can be derived.
* ****Weighting Schemes****: Techniques like TF-IDF and PMI not only adjust for term frequency but also account for the term's distribution across the corpus to highlight terms that are particularly informative about document or word meanings.
* ****Matrix Smoothing****: Methods such as SVD (Singular Value Decomposition) reduce noise and dimensionality in the data, revealing underlying semantic structures and relationships that may not be apparent in the raw frequency data.
* ****Similarity Measurement****: The cosine similarity metric is commonly used to quantify the semantic closeness between vectors. It measures the cosine of the angle between two vectors, with a higher cosine value indicating greater similarity. This measure is particularly effective in high-dimensional spaces typical of VSMs, where geometric relationships are indicative of semantic relationships.

#### 6. Applications of VSMs

* ****Enhancing Search Engines****: By representing queries and documents as vectors, search engines can rank results based on semantic similarity to the query, improving the relevance of search results.
* ****Semantic Text Analysis****: VSMs facilitate deeper analyses of text, such as identifying themes, extracting semantic relationships, and detecting sentiment, by quantifying and comparing the semantic content of text units.
* ****Natural Language Understanding****: In tasks like machine translation and speech recognition, VSMs contribute to models that can interpret and generate human-like language responses, bridging the communication gap between humans and machines.

#### 7. Future Directions and Challenges

* ****Addressing Language Complexity****: Ongoing research aims to refine VSMs to better capture the subtleties and complexities of human language, including idiomatic expressions, metaphors, and cultural nuances.
* ****Scalability and Efficiency****: As text corpora grow in size and complexity, developing more scalable and efficient VSMs remains a critical challenge, requiring innovations in algorithmic efficiency and computational resources.
* ****Cross-lingual and Multimodal Extensions****: Expanding VSMs to work across different languages and to integrate with other data modalities (e.g., visual data) are promising directions that could lead to more versatile and powerful semantic models.

### **Conclusion**

Vector Space Models of Semantics provide a powerful framework for processing and understanding text data, transforming the qualitative nuances of language into quantitative, analyzable forms. By enabling machines to approximate human-like understanding of text, VSMs have become indispensable tools in the field of natural language processing, driving forward innovations in information retrieval, content analysis, and AI-driven language technologies. As the field progresses, further advancements in VSMs will continue to enhance our ability to extract meaning from the vast expanses of textual information, opening new frontiers in computational linguistics and beyond.

For those interested in delving deeper into the intersection of AI and linguistic research, resources like [AI Scholars](https://awesomegpts.vip/" \t "/home/giorgio/Documents\\x/_new) and [Data Analysis Techniques](https://awesomegpts.vip/" \t "/home/giorgio/Documents\\x/_new) offer valuable insights and guidance on the latest methodologies and applications in the field.

Practical Example:

The provided code outlines a Python class for a K-Nearest Neighbors (KNN) classifier tailored for Natural Language Classification (NLC), leveraging the Natural Language Toolkit (NLTK) and WordNet for semantic analysis. Let's break down the code into its components for a detailed understanding.

### **Overview**

* ****Objective****: To classify text data (e.g., weather-related questions) into categories (e.g., 'temperature' or 'conditions') using a KNN algorithm based on semantic similarity of the documents.
* ****Key Components****: NLTK for text processing, WordNet for semantic similarity, pandas for data handling, and numpy for numerical operations.

### **Detailed Explanation**

#### 1. ****Setup and Preprocessing****

* ****NLTK and WordNet****: Import necessary libraries and download NLTK corpora (**genesis**, **wordnet**, **punkt** for tokenization, **averaged\_perceptron\_tagger** for part-of-speech tagging).
* ****Stopwords and Lemmatization****: Remove common stopwords and lemmatize the text to reduce words to their base form for normalization.

#### 2. ****KNN\_NLC\_Classifier Class****

* ****Initialization (**\_\_init\_\_**)****: The class is initialized with **k** (number of neighbors) and **distance\_type** ('path' for path-based similarity or other WordNet-based metrics).
* ****Training (**fit**)****: Stores the training dataset without any model training because KNN is a lazy learner.
* ****Prediction (**predict**)****: For each test instance, it computes semantic similarity with all training instances, identifies the **k** closest neighbors, and predicts the label based on the majority label of these neighbors.

#### 3. ****Semantic Similarity Calculation****

* ****Document to Synsets (**doc\_to\_synsets**)****: Converts a document into a list of WordNet synsets (basic semantic units in WordNet) by tokenizing the text, tagging part-of-speech, and finding corresponding synsets.
* ****Similarity Score (**similarity\_score**)****: Computes a similarity score between two sets of synsets, using either path similarity (based on the shortest path in the semantic hierarchy) or other similarity measures from WordNet.
* ****Document Similarity (**document\_similarity**)****: Calculates the average similarity score between two documents, treating the similarity as symmetrical.

#### 4. ****Data Loading and Preprocessing****

* Loads a dataset, renames columns for clarity, and preprocesses the text by removing punctuation, converting to lowercase, removing stopwords, and lemmatizing.

#### 5. ****Training the Classifier****

* The classifier is instantiated with **k=1** and 'path' similarity. It is then trained on the preprocessed training data.

#### 6. ****Prediction and Evaluation****

* Preprocesses a set of test questions in the same manner as the training data.
* Uses the trained classifier to predict the category (e.g., 'temperature' vs. 'conditions') for each test question.
* Outputs predictions, demonstrating the classifier's ability to categorize natural language queries based on semantic content.

### **Conclusion**

This code demonstrates a practical application of semantic analysis in NLP, using WordNet's rich lexical database for measuring document similarity in a KNN classification context. It showcases how linguistic features and semantic similarity can be harnessed to build classifiers that understand the meaning behind text data, with potential applications in areas such as intent recognition, topic classification, and question answering systems.

import nltk

from nltk.corpus import genesis

from nltk.corpus import wordnet as wn

import numpy as np

import pandas as pd

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords

nltk.download('genesis')

nltk.download('wordnet')

nltk.download('punkt')

nltk.download('averaged\_perceptron\_tagger')

genesis\_ic = wn.ic(genesis, False, 0.0)

class KNN\_NLC\_Classifer():

def \_\_init\_\_(self, k=1, distance\_type='path'):

self.k = k

self.distance\_type = distance\_type

# This function is used for training

def fit(self, x\_train, y\_train):

self.x\_train = x\_train

self.y\_train = y\_train

# This function runs the K(1) nearest neighbour algorithm and

# returns the label with closest match.

def predict(self, x\_test):

self.x\_test = x\_test

y\_predict = []

for i in range(len(x\_test)):

max\_sim = 0

max\_index = 0

for j in range(self.x\_train.shape[0]):

temp = self.document\_similarity(x\_test[i], self.x\_train[j])

if temp > max\_sim:

max\_sim = temp

max\_index = j

y\_predict.append(self.y\_train[max\_index])

return y\_predict

def convert\_tag(self, tag):

"""Convert the tag given by nltk.pos\_tag to the tag used by wordnet.synsets"""

tag\_dict = {'N': 'n', 'J': 'a', 'R': 'r', 'V': 'v'}

try:

return tag\_dict[tag[0]]

except KeyError:

return None

def doc\_to\_synsets(self, doc):

"""

Returns a list of synsets in document.

Tokenizes and tags the words in the document doc.

Then finds the first synset for each word/tag combination.

If a synset is not found for that combination it is skipped.

Args:

doc: string to be converted

Returns:

list of synsets

"""

tokens = word\_tokenize(doc + ' ')

l = []

tags = nltk.pos\_tag([tokens[0] + ' ']) if len(tokens) == 1 else nltk.pos\_tag(tokens)

for token, tag in zip(tokens, tags):

syntag = self.convert\_tag(tag[1])

syns = wn.synsets(token, syntag)

if (len(syns) > 0):

l.append(syns[0])

return l

def similarity\_score(self, s1, s2, distance\_type='path'):

"""

Calculate the normalized similarity score of s1 onto s2

For each synset in s1, finds the synset in s2 with the largest similarity value.

Sum of all of the largest similarity values and normalize this value by dividing it by the

number of largest similarity values found.

Args:

s1, s2: list of synsets from doc\_to\_synsets

Returns:

normalized similarity score of s1 onto s2

"""

s1\_largest\_scores = []

for i, s1\_synset in enumerate(s1, 0):

max\_score = 0

for s2\_synset in s2:

if distance\_type == 'path':

score = s1\_synset.path\_similarity(s2\_synset, simulate\_root=False)

else:

score = s1\_synset.wup\_similarity(s2\_synset)

if score != None:

if score > max\_score:

max\_score = score

if max\_score != 0:

s1\_largest\_scores.append(max\_score)

mean\_score = np.mean(s1\_largest\_scores)

return mean\_score

def document\_similarity(self, doc1, doc2):

"""Finds the symmetrical similarity between doc1 and doc2"""

synsets1 = self.doc\_to\_synsets(doc1)

synsets2 = self.doc\_to\_synsets(doc2)

return (self.similarity\_score(synsets1, synsets2) + self.similarity\_score(synsets2, synsets1)) / 2

# 1. Importing the dataset

# we'll use the demo dataset available at Watson NLC Classifier Demo.

# FILENAME = "https://raw.githubusercontent.com/watson-developer-cloud/natural-language-classifier-nodejs/master/training/weather\_data\_train.csv"

FILENAME = "primeri.csv"

dataset = pd.read\_csv(FILENAME, header=None)

dataset.rename(columns={0: 'text', 1: 'answer'}, inplace=True)

dataset['output'] = np.where(dataset['answer'] == 'temperature', 1, 0)

Num\_Words = dataset.shape[0]

print(dataset.head())

print("\nSize of input file is ", dataset.shape)

print("Examples: ")

print(dataset)

import re

nltk.download('stopwords')

s = stopwords.words('english')

# add additional stop words

s.extend(['today', 'tomorrow', 'outside', 'out', 'there', 'much', 'many'])

ps = nltk.wordnet.WordNetLemmatizer()

for i in range(dataset.shape[0]):

review = re.sub('[^a-zA-Z]', ' ', dataset.loc[i, 'text'])

review = review.lower()

review = review.split()

review = [ps.lemmatize(word) for word in review if not word in s]

review = ' '.join(review)

dataset.loc[i, 'text'] = review

X\_train = dataset['text']

y\_train = dataset['output']

print("Below is the sample of training text after removing the stop words")

print(dataset['text'])

# 4. Train the Classifier

classifier = KNN\_NLC\_Classifer(k=1, distance\_type='path')

classifier.fit(X\_train, y\_train)

final\_test\_list = ['will it rain', 'Is it hot outside?', 'What is the expected high for today?',

'Will it be foggy tomorrow?', 'Should I prepare for sleet?',

'Will there be a storm today?', 'do we need to take umbrella today',

'will it be wet tomorrow', 'is it humid tomorrow', 'what is the precipitation today',

'is it freezing outside', 'is it cool outside', "are there strong winds outside",

"will it be rainy tomorrow"

]

test\_corpus = []

for i in range(len(final\_test\_list)):

review = re.sub('[^a-zA-Z]', ' ', final\_test\_list[i])

review = review.lower()

review = review.split()

review = [ps.lemmatize(word) for word in review if not word in s]

review = ' '.join(review)

test\_corpus.append(review)

y\_pred\_final = classifier.predict(test\_corpus)

output\_df = pd.DataFrame(data={'text': final\_test\_list, 'code': y\_pred\_final, 'lemmatized\_text': test\_corpus})

output\_df['answer'] = np.where(output\_df['code'] == 1, 'Temperature', 'Conditions')

print("Predictions: ")

print(output\_df)