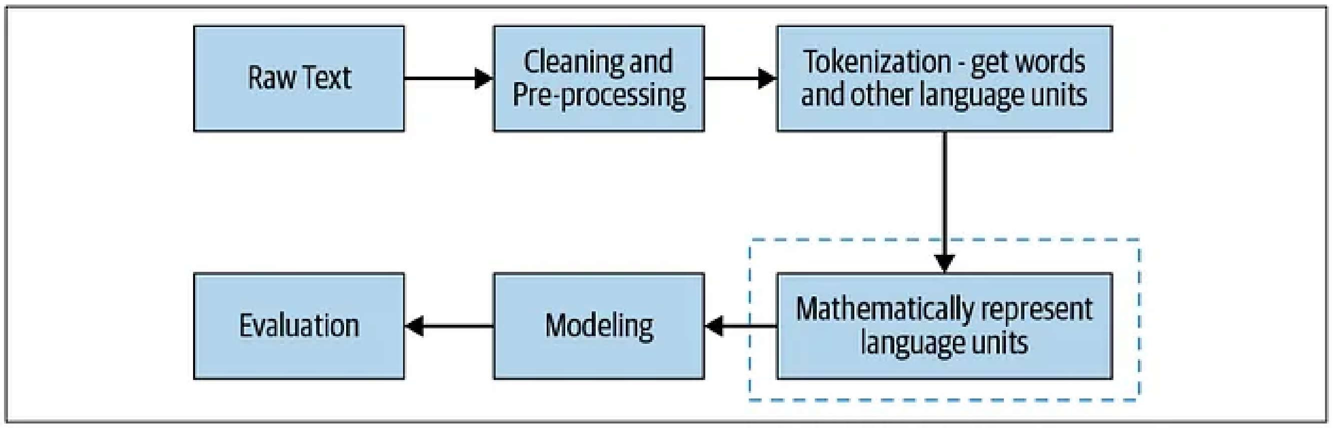
Feature Engineering in NLP

Abdallah Ashraf · Following
16 min read · Sep 13

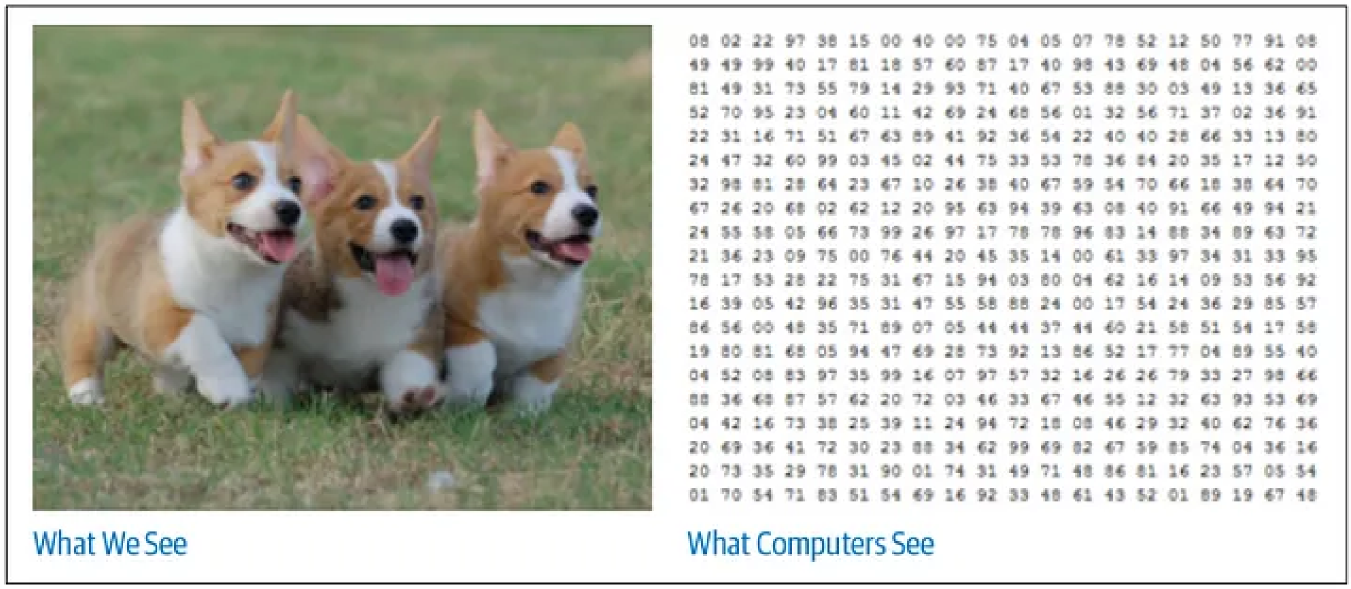
Feature extraction is an important step for any machine learning problem. No matter how good a modeling algorithm you use, if you feed in poor features, you will get poor results. In computer science, this is often called “garbage in, garbage out.”

we’ll address the question: how do we go about doing feature engineering for text data? In other words, how do we transform a given text into numerical form so that it can be fed into NLP and ML algorithms? In NLP parlance, this conversion of raw text to a suit- able numerical form is called text representation.

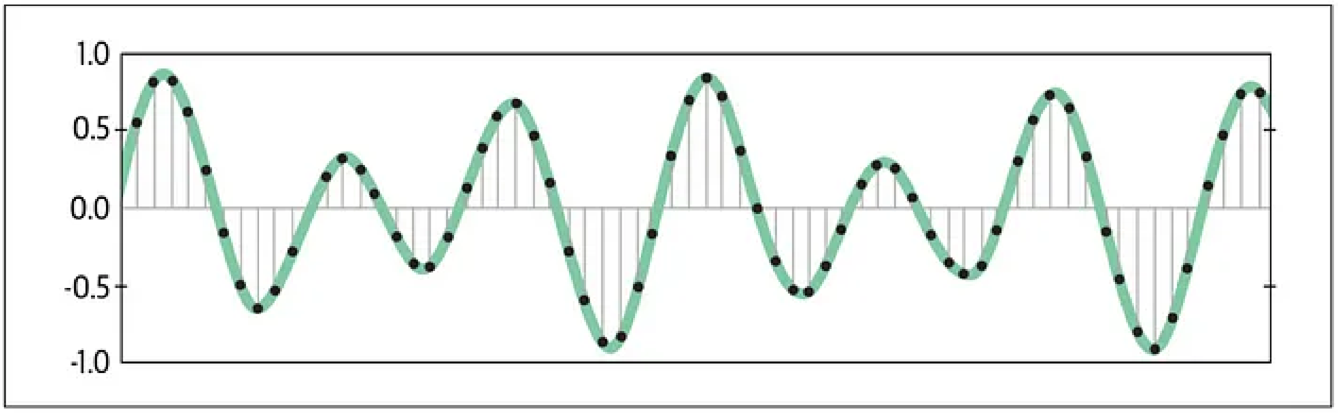
we’ll take a look at the different methods for text representation, or representing text as a numeric vector. With respect to the larger picture for any NLP problem.


Raw Text
Cleaning and Pre-processing
Tokenization - get words and other language units
Mathematically represent language units
Modeling
Evaluation

Feature representation is a common step in any ML project, whether the data is text, images, videos, or speech. However, feature representation for text is often much more involved as compared to other formats of data. To understand this, let's look at a few examples of how other data formats can be represented numerically. First, con- sider the case of images. Say we want to build a classifier that can distinguish images of cats from images of dogs. Now, in order to train an ML model to accomplish this task, we need to feed it (labeled) images. How do we feed images to an ML model? The way an image is stored in a computer is in the form of a matrix of pixels where each cell[i,j] in the matrix represents pixel i,j of the image. The real value stored at cell[i,j] represents the intensity of the corresponding pixel in the image, as shown in the figure. This matrix representation accurately represents the complete image. Video is similar: a video is just a collection of frames where each frame is an image. Hence, any video can be represented as a sequential collection of matrices, one per frame, in the same order.

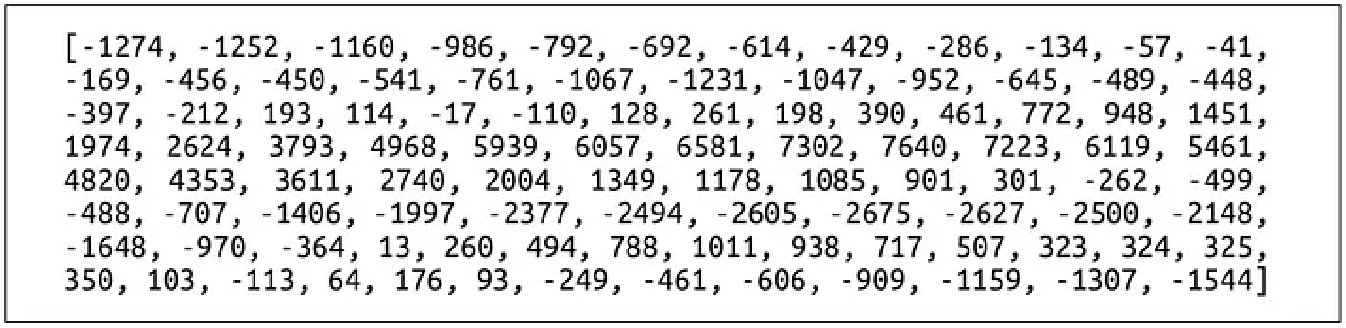


Now consider speech — it's transmitted as a wave. To represent it mathematically, we sample the wave and record its amplitude (height), as shown in the figure.



Sampling a speech wave

This gives us a numerical array representing the amplitude of a sound wave at fixed time intervals, as shown in the figure.

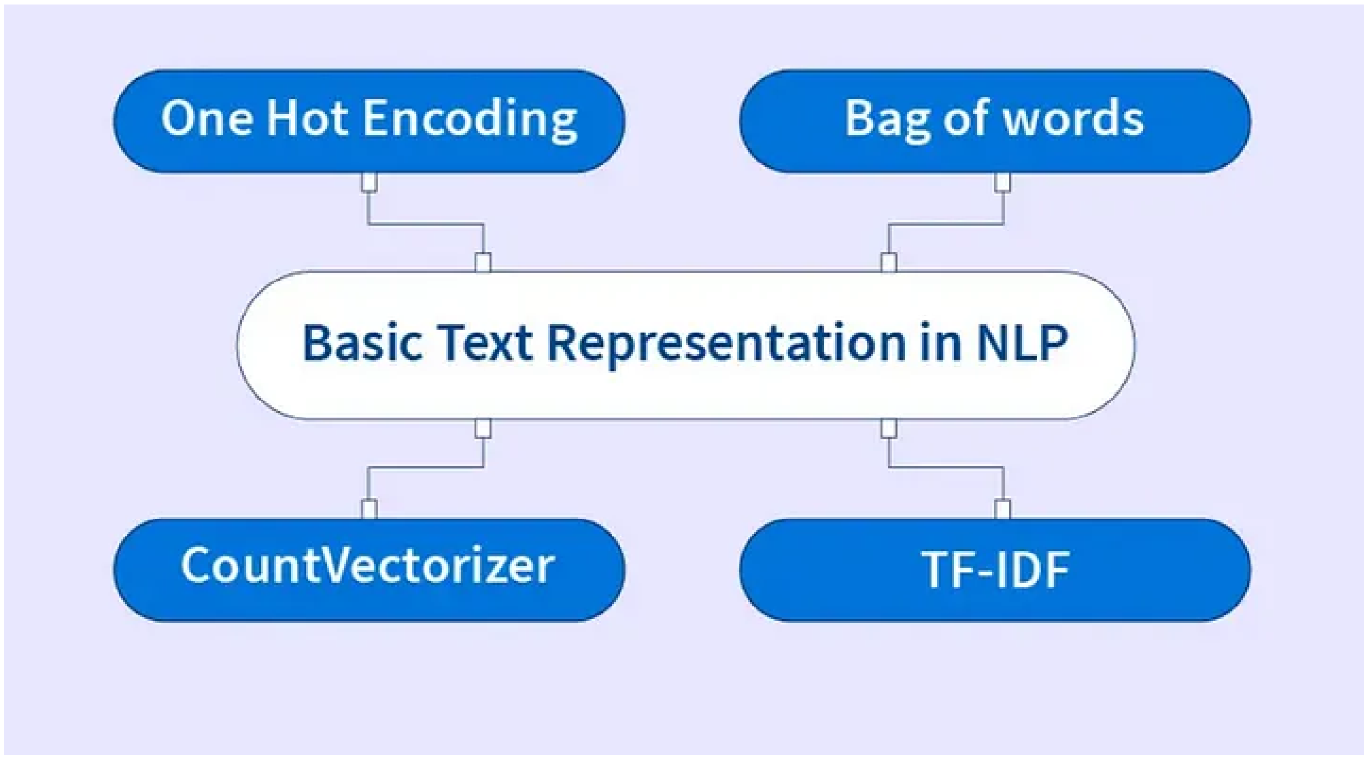


```
[-1274, -1252, -1160, -986, -792, -692, -614, -429, -286, -134, -57, -41,
-169, -456, -450, -541, -761, -1067, -1231, -1047, -952, -645, -489, -448,
-397, -212, 193, 114, -17, -110, 128, 261, 198, 390, 461, 772, 948, 1451,
1974, 2624, 3793, 4968, 5939, 6057, 6581, 7302, 7640, 7223, 6119, 5461,
4820, 4353, 3611, 2740, 2004, 1349, 1178, 1085, 901, 301, -262, -499,
-488, -707, -1406, -1997, -2377, -2494, -2605, -2675, -2627, -2500, -2148,
-1648, -970, -364, 13, 260, 494, 788, 1011, 938, 717, 507, 323, 324, 325,
350, 103, -113, 64, 176, 93, -249, -461, -606, -909, -1159, -1307, -1544]
```

From this discussion, it's clear that mathematically representing images, video, and speech is straightforward. What about text? It turns out that representing text is not straightforward, hence a whole chapter focusing on various schemes to address this question. We're given a piece of text, and we're asked to find a scheme to represent it mathematically. In literature, this is called text representation. Text representation has been an active area of research in the past decades, especially the last one. In this chapter, we'll start with simple approaches and go all the way to state-of-the-art techniques for representing text. These approaches are classified into four categories:

- Basic vectorization approaches
- Distributed representations
- Universal language representation
- Handcrafted features

This article discusses the **Basic vectorization approaches**, covering:



- One-Hot Encoding
- Bag of Words
- Bag of N-Grams
- TF-IDF

Before we delve deeper into various schemes, consider the following scenario: we're given a labeled text corpus and asked to build a sentiment analysis model. To correctly predict the sentiment of a sentence, the model needs to under- stand the meaning of the sentence. In order to correctly extract the meaning of the sentence, the most crucial data points are:

1. Break the sentence into lexical units such as lexemes, words, and phrases
2. Derive the meaning for each of the lexical units
3. Understand the syntactic (grammatical) structure of the sentence
4. Understand the context in which the sentence appears

The semantics (meaning) of the sentence arises from the combination of the above points. Thus, any good text representation scheme must facilitate the extraction of those data points in the best possible way to reflect the linguistic properties of the text. Without this, a text representation scheme isn't of much use.

Let's take a look at a key concept that carries out this process:

Vector Space Models

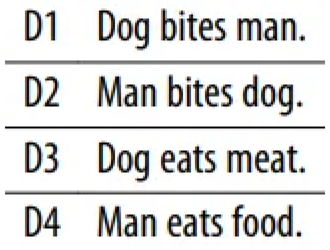
It should be clear from the introduction that, in order for ML algorithms to work with text data, the text data must be converted into some mathematical form. we'll represent text units (characters, phonemes, words, phrases, sentences, paragraphs, and documents) with vectors of numbers. This is known as the vector space model (VSM). It's a simple algebraic model used extensively for representing any text blob. VSM is fundamental to many information-retrieval operations, from scoring documents on a query to document classification and document clustering . It's a mathematical model that represents text units as vectors. In the simplest form, these are vectors of identifiers, such as index numbers in a corpus vocabulary. In this setting, the most common way to calculate similarity between two text blobs is using cosine similarity: the cosine of the angle between their corresponding vectors. The cosine of 0° is 1 and the cosine of 180° is –1, with the cosine monotonically decreasing from 0° to 180°. Given two vectors, A and B, each with n components, the similarity between them is computed as follows:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{||\mathbf{A}||_2||\mathbf{B}||_2} = \frac{\sum_{i=1}^{n} A_iB_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

where Ai and Bi are the i th components of vectors A and B, respectively. Sometimes, people also use Euclidean distance between vectors to capture similarity. All the text representation schemes we'll study fall within the scope of vector space models. What differentiates one scheme from another is how well the resulting vector captures the linguistic properties of the text it represents. With this, we're ready to discuss various text representation schemes.

Basic Vectorization Approaches

Let's start with a basic idea of text representation: map each word in the vocabulary (V) of the text corpus to a unique ID (integer value), then represent each sentence or document in the corpus as a V-dimensional vector. How do we operationalize this idea? To understand this better, let's take a toy corpus (shown in the Table) with only four documents — D1 , D2 , D3 , D4 — as an example.



D1	Dog bites man.
D2	Man bites dog.
D3	Dog eats meat.
D4	Man eats food.

Lowercasing text and ignoring punctuation, the vocabulary of this corpus is comprised of six words: [dog, bites, man, eats, meat, food]. We can organize the vocabulary in any order. In this example, we simply take the order in which the words appear in the corpus. Every document in this corpus can now be represented with a vector of size six. We'll discuss multiple ways in which we can do this. We'll assume that the text is already pre-processed (lowercased, punctuation removed, etc.) and tokenized (text string split into tokens), following the pre-processing step in the NLP pipeline. We'll start with one-hot encoding.

In one-hot encoding, each word w in the corpus vocabulary is given a unique integer ID wid that is between 1 and |V|, where V is the set of the corpus vocabulary. Each word is then represented by a V-dimensional binary vector of 0s and 1s. This is done via a |V| dimension vector filled with all 0s barring the index, where index = wid. At this index, we simply put a 1. The representation for individual words is then combined to form a sentence representation.

Let's understand this via our toy corpus. We first map each of the six words to unique IDs: dog = 1, bites = 2, man = 3, meat = 4 , food = 5, eats = 6.ii Let's consider the document D1: "dog bites man". As per the scheme, each word is a six-dimensional vector. Dog is represented as [1 0 0 0 0 0], as the word "dog" is mapped to ID 1. Bites is represented as [0 1 0 0 0 0], and so on and so forth. Thus, D1 is represented as [[1 0 0 0 0 0] [0 1 0 0 0 0] [0 0 1 0 0 0]]. D4 is represented as [[0 0 1 0 0] [0 0 0 0 1 0] [0 0 0 0 0 1]]. Other documents in the corpus can be represented similarly.

Let's look at a simple way to implement this in Python from first principles.

Since we assume that the text is tokenized, we can just split the text on white space in this example:

```
def get_onehot_vector(somestring):
  onehot_encoded = []
    for word in somestring.split():
      temp = [0]*len(vocab)
      if word in vocab:
        temp[vocab[word]-1] = 1
      onehot_encoded.append(temp)
 return onehot_encoded

get_onehot_vector(processed_docs[1])
```

```
Output: [[0, 0, 1, 0, 0, 0], [0, 1, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0]]
```

Now that we understand the scheme, let's discuss some of its pros and cons. On the positive side, one-hot encoding is intuitive to understand and straightforward to implement. However, it suffers from a few shortcomings:

- The size of a one-hot vector is directly proportional to size of the vocabulary, and most real-world corpora have large vocabularies. This results in a sparse representation where most of the entries in the vectors are zeroes, making it computationally inefficient to store, compute with, and learn from (sparsity leads to overfitting).
- This representation does not give a fixed-length representation for text, i.e., if a text has 10 words, you get a longer representation for it as compared to a text with 5 words. For most learning algorithms, we need the feature vectors to be of the same length.
- It treats words as atomic units and has no notion of (dis)similarity between words. For example, consider three words: run, ran, and apple. Run and ran have similar meanings as opposed to run and apple. But if we take their respective vectors and compute Euclidean distance between

them, they're all equally apart. Thus, semantically, they're very poor at capturing the meaning of the word in relation to other words.

- Say we train a model using our toy corpus. At runtime, we get a sentence: "man eats fruits." The training data didn't include "fruit" and there's no way to represent it in our model. This is known as the out of vocabulary (OOV) problem. A one-hot encoding scheme cannot handle this. The only way is to retrain the model: start by expanding the vocabulary, give an ID to the new word.

Some of these shortcomings can be addressed by the bag-of-words approach described next.

Bag of Words

Bag of words (BoW) is a classical text representation technique that has been used commonly in NLP, especially in text classification problems. The key idea behind it is as follows: represent the text under consideration as a bag (collection) of words while ignoring the order and context. The basic intuition behind it is that it assumes that the text belonging to a given class in the dataset is characterized by a unique set of words. If two text pieces have nearly the same words, then they belong to the same bag (class). Thus, by analyzing the words present in a piece of text, one can identify the class (bag) it belongs to.

Similar to one-hot encoding, BoW maps words to unique integer IDs between 1 and |V|. Each document in the corpus is then converted into a vector of |V| dimensions where in the ith component of the vector, i = wid, is simply the number of times the word w occurs in the document, i.e., we simply score each word in V by their occurrence count in the document.

Thus, for our toy corpus, where the word IDs are dog = 1, bites = 2, man = 3, meat = 4 , food = 5, eats = 6, D1 becomes [1 1 1 0 0 0]. This is because the first three words in the vocabulary appeared exactly once in D1, and the last three did not appear at all. D4 becomes [0 0 1 0 1 1].

The following code shows the key parts of how we can implement BoW text representation:

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()

#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#See the BOW rep for first 2 documents
print("BoW representation for 'dog bites man': ", bow_rep[0].toarray())
print("BoW representation for 'man bites dog: ",bow_rep[1].toarray())

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':",
temp.toarray())
```

If we run this code, we'll notice that the BoW representation for a sentence like "dog and dog are friends" has a value of 2 for the dimension of the word "dog," indicating its frequency in the text. Sometimes, we don't care about the frequency of occurrence of words in text and we only want to represent whether a word exists in the text or not. Researchers have shown that such a representation without considering frequency is useful for sentiment analysis . In such cases, we just initialize CountVectorizer with the binary=True option, as shown in the following code:

```
count_vect = CountVectorizer(binary=True)
bow_rep_bin = count_vect.fit_transform(processed_docs)
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':", temp.toarray())
```

This results in a different representation for the same sentence. CountVectorizer supports both word as well as character n-grams.

Let's look at some of the advantages of this encoding:

- Like one-hot encoding, BoW is fairly simple to understand and implement.
- With this representation, documents having the same words will have their vector representations closer to each other in Euclidean space as compared to documents with completely different words. The distance between D1 and D2 is 0 as compared to the distance between D1 and D4 , which is 2. Thus, the vector space resulting from the BoW scheme captures the semantic similarity of documents. So if two documents have similar vocabulary, they'll be closer to each other in the vector space and vice versa.
- We have a fixed-length encoding for any sentence of arbitrary length.

However, it has its share of disadvantages, too:

- The size of the vector increases with the size of the vocabulary. Thus, sparsity continues to be a problem. One way to control it is by limiting the vocabulary to n number of the most frequent words.

- It does not capture the similarity between different words that mean the same thing. Say we have three documents: "I run", "I ran", and "I ate". BoW vectors of all three documents will be equally apart.
- This representation does not have any way to handle out of vocabulary words (i.e., new words that were not seen in the corpus that was used to build the vectorizer).
- As the name indicates, it is a "bag" of words — word order information is lost in this representation. Both D1 and D2 will have the same representation in this scheme.

However, despite these shortcomings, due to its simplicity and ease of implementation, BoW is a commonly used text representation scheme, especially for text classification among other NLP problems.

Bag of N-Grams

All the representation schemes we've seen so far treat words as independent units. There is no notion of phrases or word ordering. The bag-of-n-grams (BoN) approach tries to remedy this. It does so by breaking text into chunks of n contiguous words (or tokens). This can help us capture some context, which earlier approaches could not do. Each chunk is called an n-gram. The corpus vocabulary, V, is then nothing but a collection of all unique n-grams across the text corpus. Then, each document in the corpus is represented by a vector of length |V|. This vector simply contains the frequency counts of n-grams present in the document and zero for the n-grams that are not present.

To elaborate, let's consider our example corpus. Let's construct a 2-gram (a.k.a. bigram) model for it. The set of all bigrams in the corpus is as follows: {dog bites, bites man, man bites, bites dog, dog eats, eats meat, man eats,

eats food}. Then, BoN representation consists of an eight-dimensional vector for each document. The bigram representation for the first two documents is as follows: D1 : [1,1,0,0,0,0,0,0], D2 : [0,0,1,1,0,0,0,0]. The other two documents follow similarly. Note that the BoW scheme is a special case of the BoN scheme, with n=1. n=2 is called a "bigram model," and n=3 is called a "trigram model." Further, note that, by increasing the value of n, we can incorporate larger context; however, this further increases the sparsity. In NLP parlance, the BoN scheme is also called "n-gram feature selection."

The following code shows an example of a BoN representation considering 1–3 n-gram word features to represent the corpus that we've used so far. Here, we use unigram, bigram, and trigram vectors by setting ngram_range = (1,3):

```
#n-gram vectorization example with count vectorizer and uni, bi, trigrams
count_vect = CountVectorizer(ngram_range=(1,3))

#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':", temp.toarray())
```

Here are the main pros and cons of BoN:

- It captures some context and word-order information in the form of n-grams.

- Thus, resulting vector space is able to capture some semantic similarity. Documents having the same n-grams will have their vectors closer to each other in Euclidean space as compared to documents with completely different n-grams.
- As n increases, dimensionality (and therefore sparsity) only increases rapidly.
- It still provides no way to address the OOV problem.

TF-IDF

In all the three approaches we've seen so far, all the words in the text are treated as equally important — there's no notion of some words in the document being more important than others. TF-IDF, or term frequency–inverse document frequency, addresses this issue. It aims to quantify the importance of a given word relative to other words in the document and in the corpus. It's a commonly used representation scheme for information-retrieval systems, for extracting relevant documents from a corpus for a given text query.

The intuition behind TF-IDF is as follows: if a word w appears many times in a document di but does not occur much in the rest of the documents dj in the corpus, then the word w must be of great importance to the document di . The importance of w should increase in proportion to its frequency in di , but at the same time, its importance should decrease in proportion to the word's frequency in other documents dj in the corpus. Mathematically, this is captured using two quantities: TF and IDF. The two are then combined to arrive at the TF-IDF score.

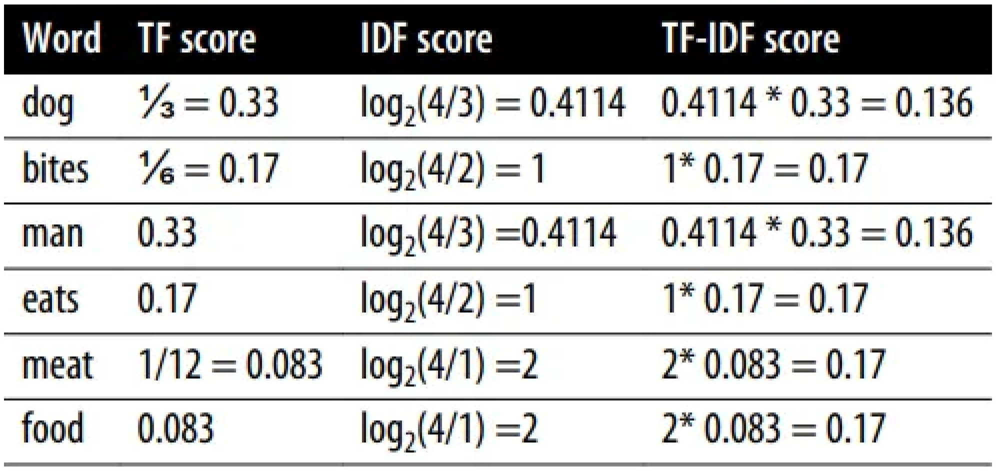
TF (term frequency) measures how often a term or word occurs in a given document. Since different documents in the corpus may be of different lengths, a term may occur more often in a longer document as compared to a shorter document. To normalize these counts, we divide the number of occurrences by the length of the document. TF of a term t in a document d is defined as:

$$\text{TF}\left(t, d\right) = \frac{(\text{Number of occurrences of term } t \text{ in document } d)}{(\text{Total number of terms in the document } d)}$$

IDF (inverse document frequency) measures the importance of the term across a cor- pus. In computing TF, all terms are given equal importance (weightage). However, it's a well-known fact that stop words like is, are, am, etc., are not important, even though they occur frequently. To account for such cases, IDF weighs down the terms that are very common across a corpus and weighs up the rare terms. IDF of a term t is calculated as follows:

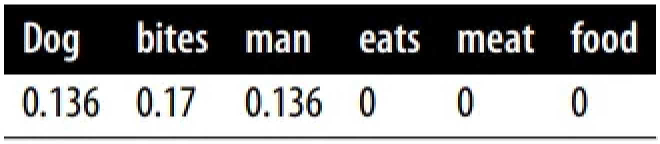
$$\text{IDF}\left(t\right) = \log_e \frac{(\text{Total number of documents in the corpus})}{(\text{Number of documents with term } t \text{ in them })}$$

The TF-IDF score is a product of these two terms. Thus, TF-IDF score = TF * IDF. Let's compute TF-IDF scores for our toy corpus. Some terms appear in only one document, some appear in two, while others appear in three documents. The size of our corpus is N=4. Hence, corresponding TF-IDF values for each term are shown in the following table .



Word	TF score	IDF score	TF-IDF score
dog	⅓ = 0.33	$\log_2(4/3) = 0.4114$	0.4114 * 0.33 = 0.136
bites	⅙ = 0.17	$\log_2(4/2) = 1$	1* 0.17 = 0.17
man	0.33	$\log_2(4/3) =0.4114$	0.4114 * 0.33 = 0.136
eats	0.17	$\log_2(4/2) =1$	1* 0.17 = 0.17
meat	1/12 = 0.083	$\log_2(4/1) =2$	2* 0.083 = 0.17
food	0.083	$\log_2(4/1) =2$	2* 0.083 = 0.17

The TF-IDF vector representation for a document is then simply the TF-IDF score for each term in that document. So, for D1 we get



Dog	bites	man	eats	meat	food
0.136	0.17	0.136	0	0	0

The following code shows how to use TF-IDF to represent text:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()
bow_rep_tfidf = tfidf.fit_transform(processed_docs)
print(tfidf.idf_) #IDF for all words in the vocabulary
print(tfidf.get_feature_names()) #All words in the vocabulary.

temp = tfidf.transform(["dog and man are friends"])
print("Tfidf representation for 'dog and man are friends':\n", temp.toarray())
```

There are several variations of the basic TF-IDF formula that are used in practice. Notice that the TF-IDF scores that we calculated for our corpus might not match the TF-IDF scores given by scikit-learn. This is because scikit-learn uses a slightly modified version of the IDF formula. This stems from provisions to account for possible zero divisions and to not entirely ignore terms that appear in all docu- ments. An interested reader can look into the TF-IDF vectorizer documentation for the exact formula.

Similar to BoW, we can use the TF-IDF vectors to calculate similarity between two texts using a similarity measure like Euclidean distance or cosine similarity. TF-IDF is a commonly used representation in application scenarios such as information retrieval and text classification. However, despite the fact that TF-IDF is better than the vectorization methods we saw earlier in terms of capturing similarities between words, it still suffers from the curse of high dimensionality.

If we look back at all the representation schemes we've discussed so far, we notice three fundamental drawbacks:

- They're discrete representations — i.e., they treat language units (words, n-grams, etc.) as atomic units. This discreteness hampers their ability to capture relation- ships between words.
- They cannot handle OOV words.
- The feature vectors are sparse and high-dimensional representations. The dimensionality increases with the size of the vocabulary, with most values being zero for any vector. This hampers learning capability. Further, high-dimensionality representation makes them computationally inefficient.

With this, we come to the end of basic vectorization approaches.

NLP Feature Engineering Data Science Artificial Intelligence

Machine Learning

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

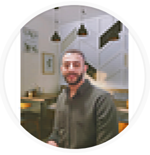
· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list



Written by Abdallah Ashraf

Following

126 Followers

Data Analytics 📈 | Data Science | Tech enthusiast | Sharing knowledge

More from Abdallah Ashraf

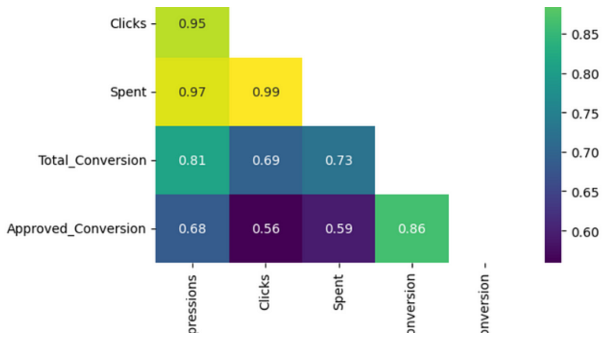
Abdallah Ashraf

Text Pre-Processing for NLP :

Let's start with a simple question: why do we still have to pre-process text?

8 min read · Aug 31

86



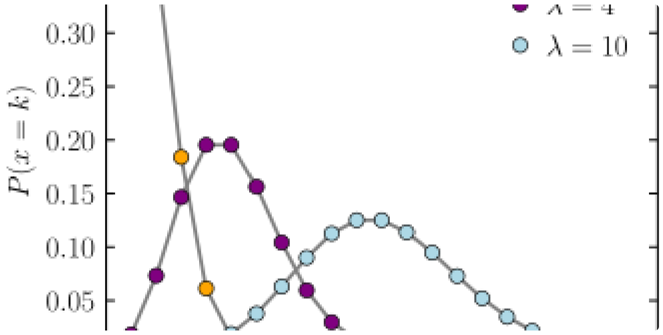
Abdallah Ashraf

Correlation in machine learning — All you need to know

What is correlation?

7 min read · Sep 22

14



Abdallah Ashraf

How to get insights from a visualization

The process of gaining insights can be understood as an effort to increase your…

6 min read · Jul 20

233 2

Abdallah Ashraf

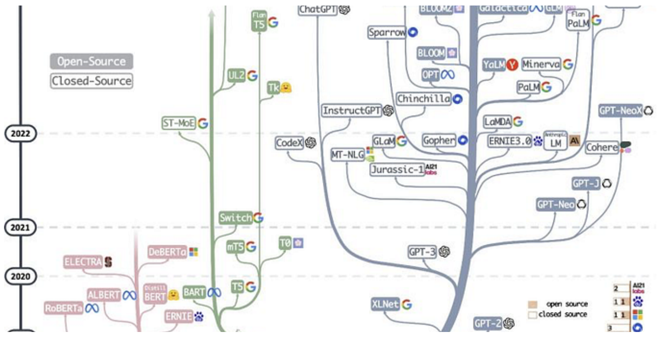
Probability Distributions— Statistics for machine learning

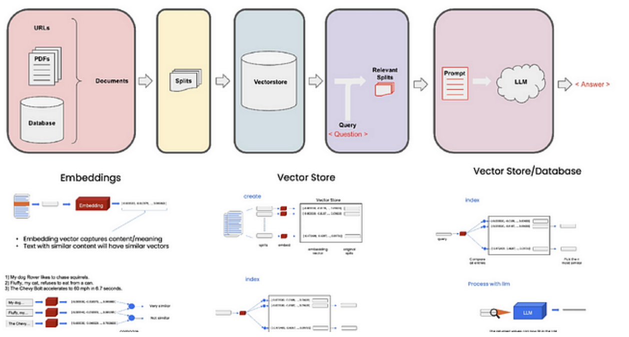
Understanding Probability distributions

9 min read · 4 days ago

See all from Abdallah Ashraf

Recommended from Medium





Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's…

15 min read · Sep 14

372

TeeTracker

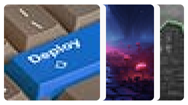
Chat with your PDF (Streamlit Demo)

Conversation with specific files

4 min read · Sep 15

56

Lists

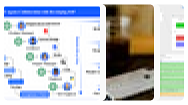


Predictive Modeling w/ Python

20 stories · 452 saves

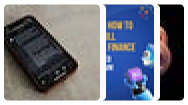
Practical Guides to Machine Learning

10 stories · 519 saves



Natural Language Processing

669 stories · 283 saves



ChatGPT prompts

24 stories · 459 saves

Alex Reed

Writing Your First NLP Python Script: From Text Preprocessing t…

Welcome to the world of Natural Language Processing (NLP)! NLP is a fascinating field a…

5 min read · Sep 22

Data Scian

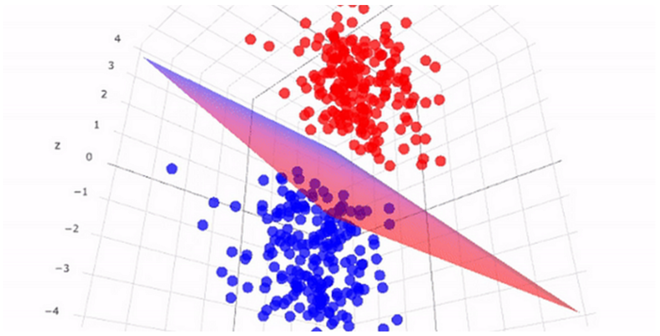
Best Portfolio Projects for Data Science

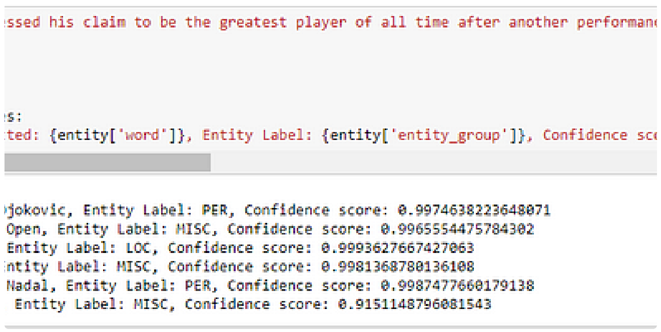
"How can I showcase my data skills to the world?" you may be asking. Fear not, for the…

5 min read · Sep 19

56

14





```
ssed his claim to be the greatest player of all time after another performan

s:
ted: {entity['word']}, Entity Label: {entity['entity_group']}, Confidence sc
```

```
jokovic, Entity Label: PER, Confidence score: 0.9974638223648071
Open, Entity Label: MISC, Confidence score: 0.9965554475784302
Entity Label: LOC, Confidence score: 0.9993627667427063
ntity Label: MISC, Confidence score: 0.9981368780136108
Nadal, Entity Label: PER, Confidence score: 0.9987477660179138
 Entity Label: MISC, Confidence score: 0.9151148796081543
```

Tasmay Pankaj Tibre... in Low Code for Data Scie...

Support Vector Machines (SVM): An Intuitive Explanation

Everything you always wanted to know about this powerful supervised ML algorithm

17 min read · Jul 1

732 4

Seffa B

Named Entity Recognition with Transformers: Extracting Metadata

3 min read · Jun 12

7

See more recommendations