Member-only story

How Does a Computer Learn to Read?

Severin Perez · Following

19 min read · 19 hours ago

89 2

Believe it or not, computers are illiterate. Despite appearances, a computer can't "read" a word, understand it's meaning, and place it in context next to adjacent words. To a computer, the words you read on the monitor are just a way of transmitting data in a way that humans can understand. The computer itself is operating on a much lower level of abstraction, because it can only store information as *bits* — ones and zeros that must be strung together to represent something more complex.

Consider the letter `a`. A computer doesn't have any way to store an `a` in memory, because computer memory is a string of switches that only have two states: on (`1`) or off (`0`). So, the computer has to find a way to represent the letter `a` with those switches. To solve this problem, humans have created various systems that match letters and symbols with decimal numbers. In ASCII for example, `a` is matched to `97`. Now, instead of the realm of letters, we're in the realm of numbers, and computers are great with numbers. A `97` can in turn be converted into binary format rather than decimal format, and we get `01100001`, which can be represented by the aforementioned switches in computer memory. To humans, this means `a`, but as far as the computer is concerned, letters don't exist.

Setting aside the intricacies of digital storage, let's consider why this matters in text analytics. Although we're not directly concerned with how computers store information, we *are* concerned with how we computers can understand and manipulate information. So, if we're interested in statistical analysis, or building machine learning models, then we need a way to convert texts into numbers so that we can carry out mathematical operations on them.

It's tempting to think, "Oh, great, we already have binary! We can just use that!" But there is a problem. As mentioned before, the computer doesn't attach any *meaning* to `01100001`. Moreover, a computer doesn't attach meaning to a string of binary digits, that convert into letters, that then spell the word `apple`, or `mountain`, or any other word. In analytics, the meaning matters, so we're going to have to find ways to convert text into numbers, which we can manipulate with math, but still maintain meaning. Let's take a look at a few ways to do this.

Just a Bag-o-Words

At its most basic level, a text is a collection of words. We know of course that the order of words matters, but even without context, the types and frequencies of words used in a text carry meaning. If, for example, I told you that the most common non-stopwords in a text were ‘goal’, ‘player’, ‘score’, and ‘ball’, then you could reasonably guess that the text has to do with sports. Using these frequencies, we can represent a text with a *bag-of-words* (BOW), so-called because this approach is to simply take all the words, count their frequencies, and toss them into a single collection without order. We can do this by first creating a vocabulary for the text, and then counting the frequencies of each word in the vocabulary.

```
# utility
import re
from collections import Counter

# data manipulation
import pandas as pd
import numpy as np

# visualization
import matplotlib.pyplot as plt
import seaborn as sns

def get_words(text: str) -> list[str]:
    """
    Extracts words from a text string, converting to lowercase
    and removing non-alphabetic characters.

    Args:
        text (str): The text to extract words from

    Returns:
        list[str]: A list of words
    """
    text = re.sub(r"[^\w\s]", "", text)
    words = text.lower().split()
    return words

def get_vocab(text: str) -> list[str]:
    """
```

```
    Extracts unique words from a text string, converting to lowercase
    and removing non-alphabetic characters.

    Args:
        text (str): The text to extract words from

    Returns:
        list[str]: An alphabetical list of unique words
    """
    words = get_words(text)
    vocab = list(set(words))
    vocab = sorted(vocab)
    return vocab
```

In our first step to create a BOW, we'll create a vocabulary. For the sake of simplicity, we're going to remove punctuation and make all words lowercase. We'll then represent the vocabulary as an alphabetically sorted list. The reason it is sorted is that we need to maintain word order to match with our frequency list.

```
animal_sentences = [
    "The cat licked his lips loudly, staring hungrily at the bird.",
    "The dog barked while the cat ate all the fish.",
    "The bird chirped loudly as the cat purred beside it.",
    "The bird on the fence watched the growling dog.",
    "With a swift leap, the cat landed from the wall."
]
animal_text = " ".join(animal_sentences)

animal_vocab = get_vocab(animal_text)
print("Length of animal_vocab:", len(animal_vocab))
print(animal_vocab)
```

```
Length of animal_vocab: 32
['a', 'all', 'as', 'at', 'ate', 'barked', 'beside', 'bird', 'cat', 'chirped', 'd
```

Now that we have a sorted list of all the words in our sample sentences about animals, we can count frequencies for each of the words.

```
def get_word_frequencies(text: str, vocab: list[str]) -> list[int]:
    """
    Counts the frequency of each word in a text string.

    Args:
        text (str): The text to count word frequencies in
        vocab (list[str]): A list of unique words to count

    Returns:
        list[int]: A list of word frequencies
    """
    words = get_words(text)
    word_counts = Counter(words)
    frequencies = [word_counts[word] for word in vocab]
    return frequencies

animal_frequencies = get_word_frequencies(animal_text, animal_vocab)
print("Length of animal_frequencies:", len(animal_frequencies))
print(animal_frequencies)
```

```
Length of animal_frequencies: 32
[1, 1, 1, 1, 1, 1, 1, 3, 4, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1
```

Note that our `animal_vocab` and `animal_frequencies` are lists of the same length. This is important because we expect the frequencies to match to the corresponding item in the vocab. To check how often a particular word is used, we should make a few helpers that convert indices to words, and words to indices.

```
# string to index map
animal_stoi = {word: i for i, word in enumerate(animal_vocab)}

# index to string map
animal_itos = {i: word for i, word in enumerate(animal_vocab)}

print(f"Word at index 10: {animal_itos[10]}")
print(f"Index of word 'dog': {animal_stoi['dog']}")

# OUTPUT
# Word at index 10: dog
# Index of word 'dog': 10
```

Now that we have our maps, we can check the frequency of particular words.

```
word = "cat"
word_index = animal_stoi[word]
word_frequency = animal_frequencies[word_index]
print(f"Frequency of '{word}' (index: {word_index}): {word_frequency}")

# OUTPUT
# Frequency of 'cat' (index: 8): 4
```

In this approach, we have stored our vocabulary and frequencies separately. We've done this for reasons that will soon become clear as we compare different text representations, but more typically we would represent a BOW as a dictionary.

```
animal_bow = {word: freq for word, freq in zip(animal_vocab, animal_frequencies)}
print("Length of animal_bow:", len(animal_bow))
print(animal_bow)
```

```
Length of animal_bow: 32
{'a': 1, 'all': 1, 'as': 1, 'at': 1, 'ate': 1, 'barked': 1, 'beside': 1, 'bird':
```

Aside from identifying word frequencies (which is certainly useful in its own right), what can we do with a BOW? Well, for starters, we can compare texts by preparing counts for all of the documents and arranging them in a single matrix. This is known as a *term-document matrix* (TDM).

```
def get_tdm(texts: list[str], vocab: list[str]) -> np.ndarray:
    """
    Creates a term-document matrix from a list of text strings.

    Args:
        texts (list[str]): A list of text strings
        vocab (list[str]): A list of unique words

    Returns:
        np.ndarray: A term-document matrix
    """
    # start with a matrix of all zeros where columns are words
    # in the shared vocab, and rows are documents
    tdm = np.zeros((len(texts), len(vocab)), dtype=int)

    # get the word frequencies for individual texts and store
    # them in the matrix
    for i, text in enumerate(texts):
        frequencies = get_word_frequencies(text, vocab)
        tdm[i] = frequencies
    return tdm

animal_tdm = get_tdm(animal_sentences, animal_vocab)
print("Shape of animal_tdm:", animal_tdm.shape)
print(animal_tdm)
```

```
Shape of animal_tdm: (5, 32)
[[0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 0 1 0 2 0 0 0 0]
```

```
 [0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 1 0]
 [0 0 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 2 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 3 0 1 0 0]
 [1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 2 1 0 0 1]]
```

```
# test that the column sums (total word counts across all documents)
# match the word frequencies in the vocabulary
column_sums = np.sum(animal_tdm, axis=0)
match = np.array_equal(column_sums, animal_frequencies)

print("Column sums match frequencies:", match)

# OUTPUT
# Column sums match frequencies: True
```

With our TDM, we can now do all sorts of interesting statistical comparisons, but perhaps most importantly, we can use the documents as inputs to train machine learning models. Imagine we had some label for each sentence, like `[1, 0, 0, 1, 1]` where `0` means "Class A" and `1` means "Class B". Having a numerical representation of each document with a known label would allow us to train a model to classify documents. We won't get into the details of training a model here, but you can see why the numerical representation matters—without it, we wouldn't have anything to feed into the model, since models can only operate on numerical representations of text.

The BOW and TDM approach certainly has value (and if you want to prepare your own BOW, you can head over to the Lemmalytica dashboard), but we're missing something important: word order. We know that word order is critical to language, and using a BOW loses that meaning. Let's consider some ways to represent our text numerically without losing word order.

It's Getting Hot in Here...

One approach to representing a text numerically without losing the information we get from word order is *one-hot encoding*. In this scheme, we

Open in app ↗

Medium Search Write 2

```
def get_one_hot_vocab(vocab: list[str]) -> np.ndarray:
    """
    Creates a one-hot encoding matrix from a list of unique words.

    Args:
        vocab (list[str]): A list of unique words

    Returns:
        np.ndarray: A one-hot encoding matrix
    """
    vocab_size = len(vocab)
    one_hot_vocab = np.eye(vocab_size)
    return one_hot_vocab

animal_one_hot_vocab = get_one_hot_vocab(animal_vocab)

print("Shape of animal_one_hot_vocab:", animal_one_hot_vocab.shape)
animal_one_hot_vocab
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]], shape=(32, 32))
```

Here, we see the one-hot vocabulary, which matches up by index with our word vocabulary. Each row in the one-hot vocabulary consists of all zeros, except for a one in the column corresponding to the word index. We can see

this below, with the first two words from the vocabulary. Notice how the `1` moves one spot to the right with each item.

```
for i, word in enumerate(animal_vocab[:3]):
    print(f"One-hot encoding for '{word}':")
    print(animal_one_hot_vocab[i])
    print()
```

```
One-hot encoding for 'a':
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.]

One-hot encoding for 'all':
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.]

One-hot encoding for 'as':
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.]
```

Now that we have our one-hot vocabulary, we can use it to create a one-hot matrix (OHM) for our text as a whole.

```
def get_one_hot_matrix(text: str, vocab: list[str]) -> np.ndarray:
    """
    Encodes a text string using one-hot encoding.

    Args:
        text (str): The text to encode
        vocab (list[str]): A list of unique words

    Returns:
        np.ndarray: A one-hot encoded matrix
    """
    one_hot_vocab = get_one_hot_vocab(vocab)
    stoi = {word: i for i, word in enumerate(vocab)}
```

```
    words = get_words(text)
    word_indices = [stoi[word] for word in words]

    return one_hot_vocab[word_indices]

def one_hot_to_word(one_hot_row: np.ndarray, vocab: list[str]) -> str:
    """
    Converts a one-hot encoded row back to a word.

    Args:
        one_hot_row (np.ndarray): A one-hot encoded row
        vocab (list[str]): A list of unique words

    Returns:
        str: The word corresponding to the one-hot row
    """
    itos = {i: word for i, word in enumerate(vocab)}
    word_index = np.argmax(one_hot_row)
    return itos[word_index]

animal_one_hot_matrix = get_one_hot_matrix(animal_text, animal_vocab)
print("Shape of animal_one_hot_matrix:", animal_one_hot_matrix.shape)
print(animal_one_hot_matrix)
```

```
Shape of animal_one_hot_matrix: (50, 32)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Looking at the shape of our one-hot matrix (OHM) for the animal text, we see that it is `(50, 32)`, meaning 50 rows by 32 columns. This is exactly what we expect, given that there are 50 total words in the text, and 32 unique vocabulary words. We can confirm that our OHM is correct by checking if the rows of the OHM have the `1` in the same index as the word's vocab index.

```
for word_idx, word in enumerate(get_words(animal_text)[:5]):
    vocab_idx = animal_stoi[word]
    row = animal_one_hot_matrix[word_idx]
    one_idx = np.argmax(row)

    print(f"{word} ({vocab_idx}) ->")
    print(f"{row} ->")
    print(f"{one_idx} == {vocab_idx}")
    print("")
```

```
the (27) ->
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 1. 0. 0. 0. 0.] ->
27 == 27

cat (8) ->
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.] ->
8 == 8

licked (20) ->
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.] ->
20 == 20

his (15) ->
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.] ->
15 == 15

lips (21) ->
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.] ->
21 == 21
```

As with our BOW approach, we could now use OHM representations to feed texts into a machine learning model. But unlike the BOW, the OHM representation preserves word order, and the associated information that comes with the patterns in that order. We'll leave implementation of that to

another time, because now we have yet another problem! Although we have gained some information by preserving word order, we still don't have any information about the *meaning* of the words. For that, we're going to have to turn to one of the most important tools in machine learning and text analytics: embeddings.

Embedding Linguistic Meaning in Numbers

To say that embeddings have revolutionized machine learning is an understatement. Along with transformers, embeddings are arguably one of the most important ideas that led to the dramatic progress we have seen in machine learning and deep learning. The reason is that embeddings give us more than just abstract representations of text; they can also give us contextual meaning. To get an idea of what that means, let's explore two types of embeddings: random embeddings, and contextual embeddings.

First off, what is a text embedding? In short, it's a vector of numbers that represents a word. In a similar fashion to one-hot encoding, we can create an embedding by iterating over the vocabulary of a text and generating a vector to represent each word. However, unlike the one-hot encoding approach where the vector was all zeros, except for a one at the index of the word, in the embedding approach the numbers are all decimals (aka floats). Let's start out by creating some random embeddings.

```
def create_word_embeddings(
        vocab: list[str], embedding_dim: int = 3, seed: int = 42
    ) -> dict[str, np.ndarray]:
    """
    Create word embeddings for a vocabulary.

    Args:
        vocab (list[str]): List of words in the vocabulary
        embedding_dim (int, optional): Dimension of the embedding vectors. Defau
```

```
            seed (int, optional): Random seed for reproducibility. Defaults to 42.

        Returns:
            dict[str, np.ndarray]: Dictionary mapping words to their embedding vecto
        """
        np.random.seed(seed)
        embeddings = np.array([np.random.normal(0, 0.1, (embedding_dim,)) for word i

        return embeddings

animal_word_embeddings = create_word_embeddings(animal_vocab)

print("Shape of animal_word_embeddings:", animal_word_embeddings.shape)

# OUTPUT
# Shape of animal_word_embeddings: (32, 3)
```

As with our one-hot vocabulary, our embeddings vocabulary has 32 rows, one for each word in the word vocabulary. The number of columns though differs. In our example, we have just three columns, or three *dimensions* in our embeddings. We might just as easily increased this to 10, or 50, or 100,000 dimensions. The more dimensions, the more fine grained the representation, but it comes at a cost of efficiency and size. In practice, major language models have thousands of embeddings.

Earlier, I said that embeddings are advantageous because they can approximate *meaning* (at least in a vague sense). To understand what this means, we'll look at a few sample words, and then come back to them when we use our contextual embeddings.

```
cat_embedding = animal_word_embeddings[animal_stoi["cat"]]
dog_embedding = animal_word_embeddings[animal_stoi["dog"]]
```

```
print("Embedding for 'cat':", cat_embedding)
print("Embedding for 'dog':", dog_embedding)
```

```
Embedding for 'cat': [-0.05443827  0.01109226 -0.11509936]
Embedding for 'dog': [-0.06017066  0.18522782 -0.00134972]
```

Above are the embeddings for ‘cat’ and ‘dog’. They’re different (as expected), but do they have meaning beyond being unique? One of the amazing things about embeddings is that we can calculate the distance between them, which actually lets us measure whether they are related. We do this using a method called *cosine similarity*, which is a linear algebra operation that takes two vectors A and B (like our embeddings) and returns a value in the range [−1,1]. For the sake of simplicity, we’ll leave a full explanation for another time, but the important thing to know is that a value of −1 indicates exact opposites, a value of 0 means the vectors are orthogonal (no relation), and a value of 1 means they’re exactly the same.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Now, we can use cosine similarity to measure the distance between or dog and cat embeddings.

```
def cosine_similarity(v1: np.ndarray, v2: np.ndarray) -> float:
    """
    Calculate the cosine similarity between two vectors.

    Args:
        v1 (np.ndarray): First vector
        v2 (np.ndarray): Second vector
```

```
    Returns:
        float: Cosine similarity between the vectors (-1 to 1)
    """
    # Calculate dot product
    dot_product = np.dot(v1, v2)

    # Calculate magnitudes
    norm_v1 = np.linalg.norm(v1)
    norm_v2 = np.linalg.norm(v2)

    # Calculate cosine similarity
    similarity = dot_product / (norm_v1 * norm_v2)

    return similarity

cat_dog_similarity = cosine_similarity(cat_embedding, dog_embedding)
print("Cosine similarity between 'cat' and 'dog':", cat_dog_similarity)

# OUTPUT
# Cosine similarity between 'cat' and 'dog': 0.22037684515784212
```

For our random embeddings, we have a cosine similarity of `0.22037` for the words 'cat' and 'dog', suggesting a mild measure of relation. Is this what we expect? In the popular imagination, dogs and cats are of course opposites, but when you think about it, they're actually very similar. Both are mammals, both have four legs, both have fur, and both are human companions. Given as much, a mild relationship doesn't quite seem right. But, since our embeddings are random, this result is also random. (With a different seed, we would get something totally different!) In other words, the embeddings have no meaning whatsoever, other than differentiating between words.

Before moving on, let’s go ahead and convert our full text to embeddings, and see what comes out.

```
def text_to_embeddings(text: str, vocab: list[str], embeddings: np.ndarray) -> n
    """
    Convert text to sequence of word embeddings.

    Args:
        text (str): Input text
        vocab (list[str]): List of vocabulary words
        embeddings (np.ndarray): Array of word embeddings

    Returns:
        np.ndarray: Array of word embeddings for the text
    """
    words = get_words(text)

    # Get word to index mapping
    stoi = {word: i for i, word in enumerate(vocab)}

    # Get embedding for each word
    text_embeddings = []
    for word in words:
        word_idx = stoi[word]
        text_embeddings.append(embeddings[word_idx])

    return np.array(text_embeddings)

word_embeddings = create_word_embeddings(animal_vocab)
animal_text_embeddings = text_to_embeddings(animal_text, animal_vocab, word_embe
print("Shape of animal_text_embeddings:", animal_text_embeddings.shape)
print(animal_text_embeddings)
print()
```

```
Shape of animal_text_embeddings: (50, 3)
[[ 0.03571126  0.1477894  -0.05182702]
 [-0.05443827  0.01109226 -0.11509936]
 [-0.04791742 -0.0185659  -0.1106335 ]
 [-0.07198442 -0.04606388  0.10571222]
 [-0.11962066  0.08125258  0.135624  ]
 [-0.00720101  0.10035329  0.0361636 ]
 [ 0.08219025  0.00870471 -0.02990074]
...
 [ 0.01968612  0.07384666  0.01713683]
 [ 0.03571126  0.1477894  -0.05182702]
 [-0.08084936 -0.0501757   0.09154021]]
```

Adding Context to Embeddings

The obvious first question is, *how* do we give embeddings meaning? The answer is more complicated than we can cover in this article, but the quick version is as follows. First, we take random embeddings like the ones we created before. And then, we use them to train an embedding model, usually on a task like predicting words. On each training cycle, the model updates the weights — that is, the decimal numbers inside our embeddings. Over time, as those weights are updated, they start to acquire meaning beyond just being unique as compared with other words. We can see this by using the same cosine similarity function we looked at before.

In our example, we'll use pre-trained BERT embeddings, but there are lots of other options, like GPT, FastText, and more. The important thing is that these are not random embeddings, but *contextual embeddings* that have been trained in a way that gives them meaning.

```
from transformers import AutoTokenizer, AutoModel
import torch

def get_contextual_embeddings(
    text: str,
    model: torch.nn.Module,
    tokenizer: AutoTokenizer,
    device: torch.device,
    vocab: list[str] = None,
) -> dict[str, torch.Tensor]:
    """
    Get contextual embeddings for text.

    Args:
        text (str): Input text
        model (torch.nn.Module): Pretrained model
        tokenizer (AutoTokenizer): tokenizer
        device (torch.device): Device to run the model on
        vocab (list[str], optional): List of words to create embeddings for.
                                     If None, returns raw embeddings.
```

```
    Returns:
        dict[str, torch.Tensor]: Dictionary mapping words to their embeddings if
                                 otherwise returns raw embeddings
    """
    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
    inputs = {k: v.to(device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

    if vocab is None:
        return embeddings

    embedding_map = {}
    tokens = tokenizer.convert_ids_to_tokens(inputs['input_ids'][0])

    for word in vocab:
        word_tokens = tokenizer(word, return_tensors="pt")['input_ids'][0][1:-1]
        word_embeddings = []

        for i in range(len(tokens)):
            if tokens[i] == tokenizer.convert_ids_to_tokens(word_tokens.cpu())[0
                word_embeddings.append(embeddings[0, i].cpu())

        if word_embeddings:
            embedding_map[word] = torch.stack(word_embeddings).mean(0)

    return np.array(list(embedding_map.values()))

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
bert_tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
bert_model = AutoModel.from_pretrained("bert-base-uncased").to(device)

contextual_word_embeddings = get_contextual_embeddings(
    animal_text,
    model=bert_model,
    tokenizer=bert_tokenizer,
    device=device,
    vocab=animal_vocab
)

print(contextual_word_embeddings.shape)

# OUTPUT
# (32, 768)
```

One thing to notice in our contextual embeddings is just how big they are! Each embedding has 768 dimensions, which as you can imagine, requires a very big model when you are talking about every word in a language (rather than just our small vocabulary of 32 words in our animal texts). Now, let's compare these embeddings on the same words, 'cat' and 'dog', as we did with our random embeddings.

```
cat_contextual_embedding = contextual_word_embeddings[animal_stoi["cat"]]
dog_contextual_embedding = contextual_word_embeddings[animal_stoi["dog"]]

cat_dog_contextual_similarity = cosine_similarity(
    cat_contextual_embedding,
    dog_contextual_embedding
)

print("Cosine similarity for words 'cat' and 'dog':")
print("Random embeddings:", cat_dog_similarity)
print("Contextual embeddings:", cat_dog_contextual_similarity)
```

```
Cosine similarity for words 'cat' and 'dog':
Random embeddings: 0.22037684515784212
Contextual embeddings: 0.86056995
```

Huge difference! As we discussed before, if the embeddings have meaning, we would expect 'cat' and 'dog' to be pretty similar, and indeed, with the BERT embeddings they have a cosine similarity of 0.8605. Pretty darn close! Of course, as we said, our random embeddings *could* have shown closer similarity if the random rolls had turned out differently. Let's try a few other word pairs to confirm that the contextual embeddings really do have more meaning.

```
word_pairs = [
    ("cat", "dog"),
    ("cat", "bird"),
    ("cat", "fish"),
    ("cat", "wall"),
    ("wall", "fence"),
]

similarities = []
for w1, w2 in word_pairs:
    idx1, idx2 = animal_stoi[w1], animal_stoi[w2]

    emb1, emb2 = animal_word_embeddings[idx1], animal_word_embeddings[idx2]
    sim1 = cosine_similarity(emb1, emb2)

    emb1, emb2 = contextual_word_embeddings[idx1], contextual_word_embeddings[id
    sim2 = cosine_similarity(emb1, emb2)

    similarities.append({
        'word 1': w1,
        'word 2': w2,
        'random similarity': sim1,
        'contextual similarity': sim2
    })

df_similarities = pd.DataFrame(similarities)
display(df_similarities)
```

word 1	word 2	random similarity	contextual similarity
cat	dog	0.220377	0.860570
cat	bird	0.959157	0.766288
cat	fish	0.394307	0.400762
cat	wall	-0.396528	0.391683
wall	fence	-0.282078	0.697818

As we look at multiple word pairs, it becomes clear that contextual embeddings are working as expected. With our random embeddings, the similarity is... well.. random. But with the contextual embeddings we seem

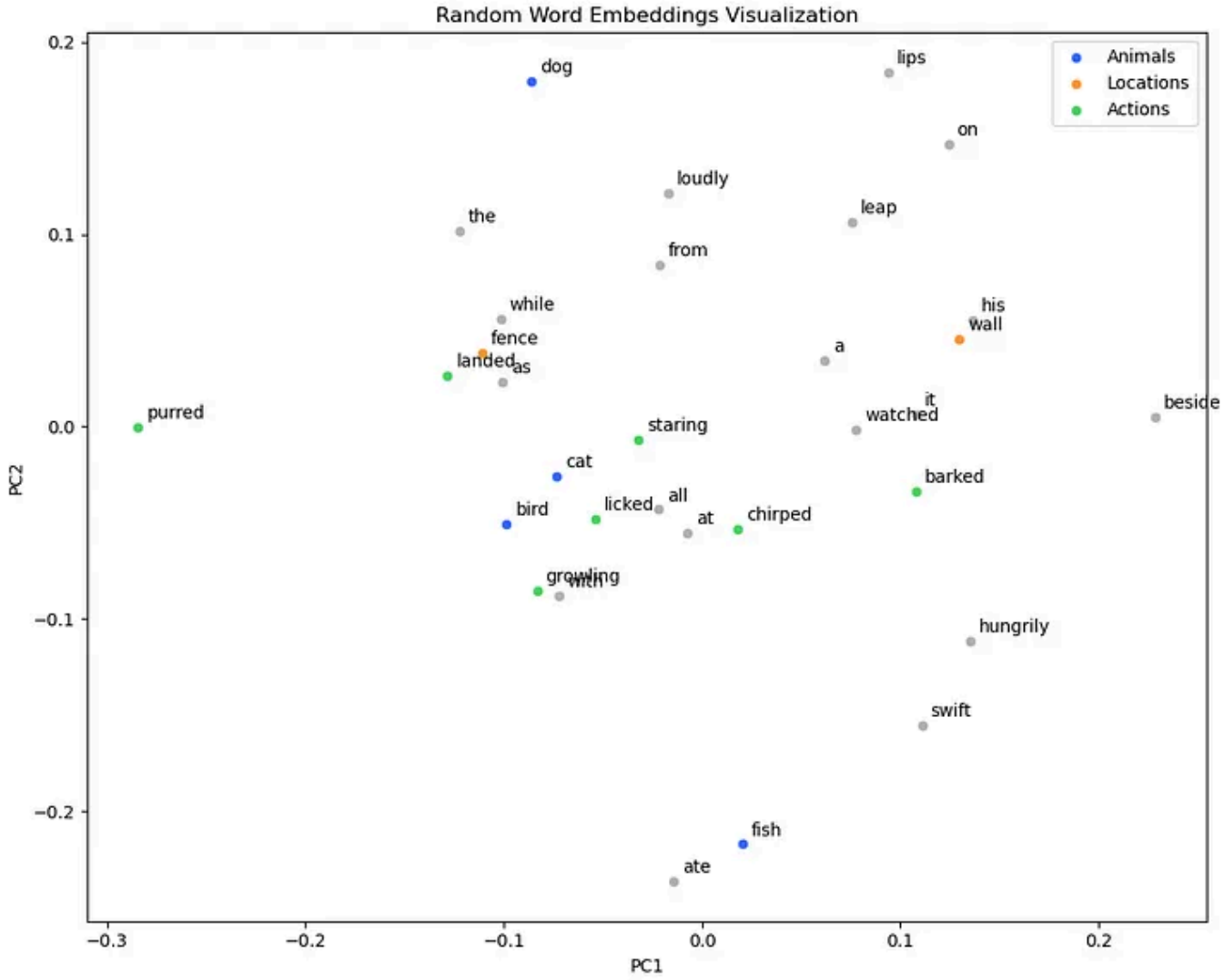
to have real meaning. The word ‘cat’ is very similar to ‘dog’ (both are mammals, furry, household pets), ‘cat’ is less similar to ‘bird’, even less similar to ‘fish’, and not really similar at all to ‘wall’. However, ‘wall’ is quite similar to ‘fence’. All of this falls within our expectations, and shows the power of contextual embeddings. To illustrate even further, we can plot our two sets of embeddings using a technique called dimensionality reduction, which let’s us plot multi-dimensional vectors (like our embeddings) on a two-dimensional canvas. (We’ll ignore the details of dimensionality reduction for now, but it’s worth looking into if you’re interested!)

```
from sklearn.decomposition import PCA

def plot_embeddings(embeddings: np.ndarray, vocab: list[str], highlight_words: l
                    figsize: tuple=(10, 8), title: str="Word Embeddings Visualiza
    """
    Plots word embeddings in 2D space with highlighted word groups.

    Args:
        embeddings (np.ndarray): Matrix of word embeddings
        vocab (list[str]): List of words corresponding to embeddings
        highlight_words (list[dict]): List of dictionaries with 'title' and 'wor
                                      for highlighting word groups
        figsize (tuple): Figure size (width, height)
        title (str): Plot title. Defaults to "Word Embeddings Visualization"
    """
    pca = PCA(n_components=2)
    embeddings_2d = pca.fit_transform(embeddings)
    df = pd.DataFrame({
        'PC1': embeddings_2d[:, 0],
        'PC2': embeddings_2d[:, 1],
        'word': vocab
    })

    plt.figure(figsize=figsize)

    highlighted = set(word for group in highlight_words for word in group['words
    regular_words = [w for w in vocab if w not in highlighted]
    regular_mask = df['word'].isin(regular_words)
    sns.scatterplot(data=df[regular_mask], x='PC1', y='PC2', alpha=0.6, color='g

    colors = sns.color_palette('bright', n_colors=len(highlight_words))
```

```
        for group, color in zip(highlight_words, colors):
            mask = df['word'].isin(group['words'])
            sns.scatterplot(data=df[mask], x='PC1', y='PC2', alpha=0.8,
                            color=color, label=group['title'])

        for idx, row in df.iterrows():
            plt.annotate(row['word'], (row['PC1'], row['PC2']),
                         xytext=(5, 5), textcoords='offset points')

        plt.title(title)
        plt.legend()
        plt.tight_layout()

    animal_text_contextual_embeddings = text_to_embeddings(
        animal_text,
        animal_vocab,
        contextual_word_embeddings
    )

    print("Shape of animal_text_contextual_embeddings:", animal_text_contextual_embe
```

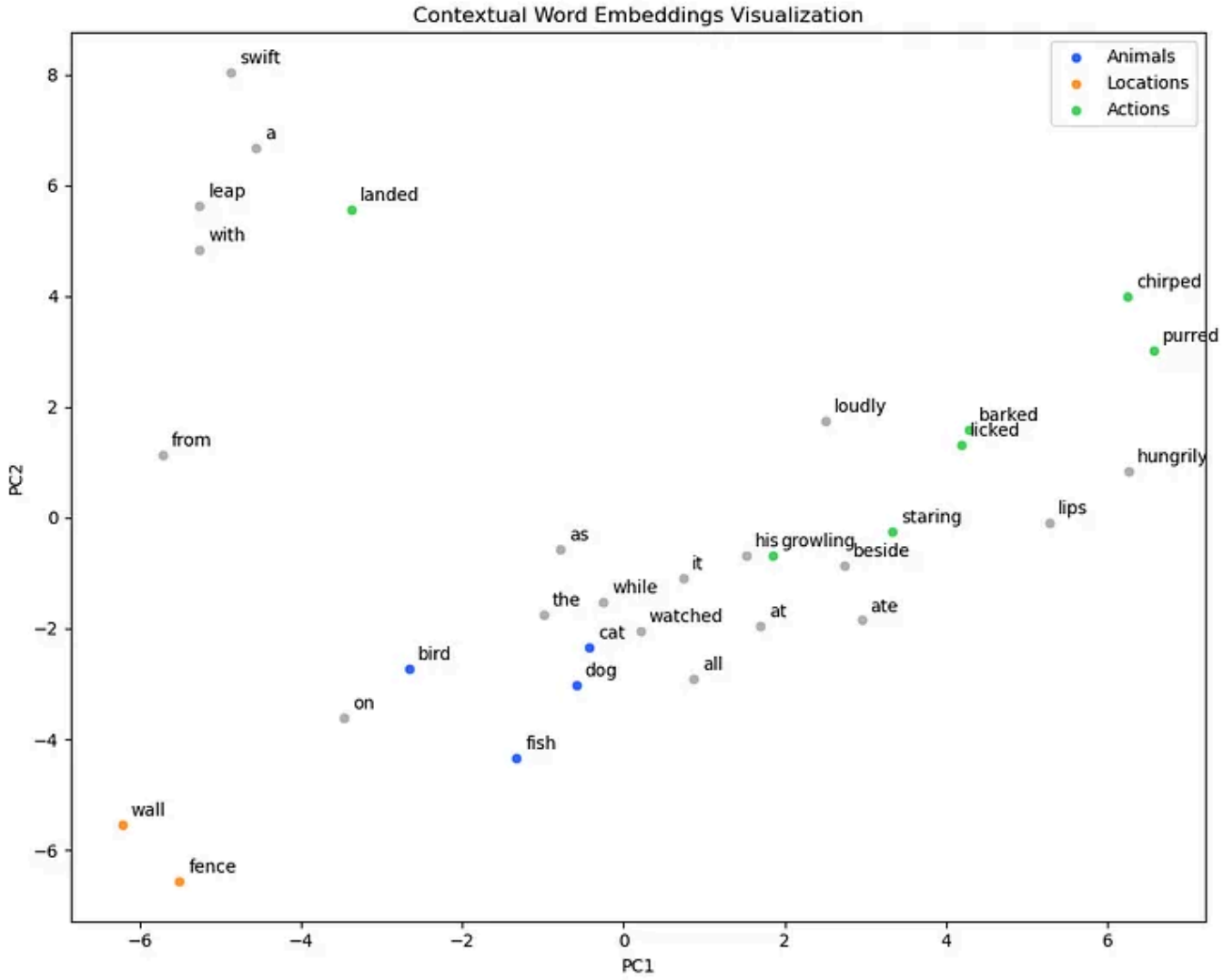
```
Shape of animal_text_contextual_embeddings: (50, 768)
```

First, let's plot the random embeddings, highlighting a few words that we expect to be similar.



In the plot of random word embeddings, the pattern seems essentially random (per expectations). Compare this to the plot of contextual embeddings.

```
plot_embeddings(
    contextual_word_embeddings,
    animal_vocab,
    highlight_words=highlight_words,
    title="Contextual Word Embeddings Visualization"
)
```



As expected from our cosine similarity calculations, here we see a much clearer pattern. The points for ‘dog’ and ‘cat are directly next to one another, as are ‘wall’ and ‘fence’. It’s not a perfect system, but when we’re trying to convert the meaning we find in words into a format a computer can process, it’s pretty darn neat!

Is the Computer Literate Now?

Converting text representations into numbers clearly has huge benefits. The computer can now process them, run mathematical calculations, build models, etc. All things that weren’t possible with pure text. But is the computer literate? At some point, we leave the realm of computer science

and enter the realm of philosophy. The computer still doesn't *understand* the meaning in our texts, but it can recognize and predict patterns. In some ways, perhaps pattern recognition and "knowledge" aren't all that different. Does the human brain, after all, not do the same thing? At some point, the distinction between how computers understand information, and how humans do, will become irrelevant. For our purposes in text analytics though, we can at least say that having ways of representing text as numbers gives us new and powerful tools.

The above article originally appeared on the Lemmalytica Blog. If you're interested in more articles, references, and tools for text analytics, check it out!

Data Science | Coding | Programming | Text Analytics

Written by Severin Perez

2.6K Followers · 27 Following

Writer | Developer

Following

Responses (2)

What are your thoughts?

Respond

Sohaib wagi he
30 mins ago

Great article

Reply

Muhammad Wasif
2 hours ago

> Setting aside the intricacies of digital storage, let's consider why this matters in text analytics

Great article keep it up 💯💯

1 reply Reply

More from Severin Perez

Severin Perez

The Five Lies

The Obscuration of Truth, in Five Ways

Mar 18, 2018 9

Severin Perez

Writing Flexible Code with the Single Responsibility Principle

SOLID Principles and Maintainable Code

Sep 7, 2018 2.9K 15

Severin Perez

Making the Most of Polymorphism with the Liskov Substitution...

Designing Subtypes in SOLID Code

Oct 4, 2018 1.5K 5

Severin Perez

Italian Aperitivo Face-off

Campari v. Cappelletti

Jun 19, 2018

See all from Severin Perez

Recommended from Medium

Rajan Sahu

Indexing in PostgreSQL Part-1

My article is for everyone! Non-members can simply click this link and jump straight into...

Jan 31 75

MB20261

Python by Examples: Mastering Pandas DataFrames (2 of 2)

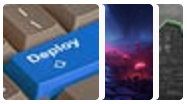
Data analysis is a crucial component of modern decision-making, and Pandas...

Jan 20

Lists

General Coding Knowledge

20 stories · 1904 saves



Predictive Modeling w/ Python

20 stories · 1818 saves

Coding & Development

11 stories · 1001 saves

Stories to Help You Grow as a Software Developer

19 stories · 1591 saves

Tom Wilson

A Critique of the Elegoo Tumbller—Useful Learning Device or STEM...

Can the Elegoo Tumbller be used as a STEM teaching tool? The answer is complicated...

Dec 27, 2024 1

LM Po

Mastering Python Classes: A Comprehensive Guide

Python, as a versatile and powerful programming language, offers developers a...

Jan 28 117

Sylvain Tiset

35 Laws and Principles in Software Development

Here are some interesting quotes, laws and principles in Software development. They ar...

4d ago 114

Ebrahim Mousavi

NLP Series: Day 4—Stopword Removal and Normalization

Refining Text Data for Enhanced Natural Language Processing Accuracy

Jan 15 102

See more recommendations