Natural Language Processing

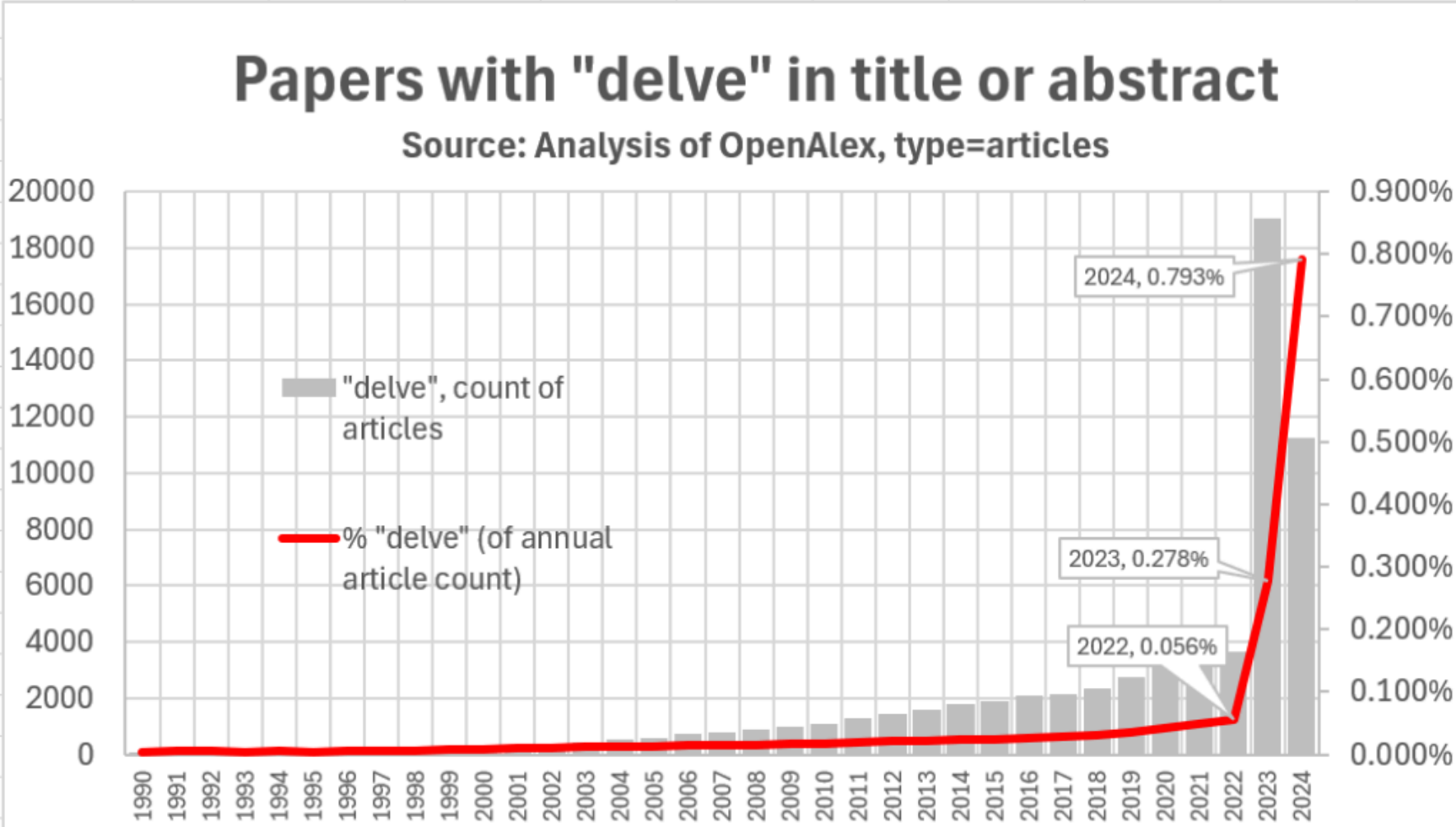
Do Now: Songs w/ Pandas

bigd103.link/pandas-lyrics

Natural Language Processing (NLP) is an interdisciplinary subfield of computer science and artificial intelligence. It is primarily concerned with enabling computers to process data encoded in natural language. NLP is closely related to information retrieval, knowledge representation, and computational linguistics—a subfield of linguistics.

Applications of NLP in Real-World Scenarios

- Language translation (e.g., Google Translate).
- Filtering internet hate speech, fake news, and spam.
- Content recommendation systems.
- *A component of* text-completion suggestions (e.g., autocomplete).
- *A component of* language processing and auto-captioning.
- *A component of* large language models (LLMs), such as ChatGPT.



Analysis using OpenAlex by Philip Shapira, March 31, 2024.

Popular NLP Libraries

NLTK:

- Natural Language Toolkit, an open-source Python library by Steven Bird, Edward Loper, and Ewan Klein.
- Very comprehensive, widely used, but somewhat older and can be confusing to learn initially.

spaCy:

- Implemented in Cython and developed by Matthew Honnibal.
- Faster, easy to use, and more modern than NLTK but somewhat less comprehensive out-of-the-box.

TextBlob:

- Built on top of NLTK and Pattern, providing a simple API for common NLP tasks.
- Great for beginners and rapid prototyping.

NLP Terminology

Document:

- A piece of text, such as a sentence, a paragraph, a song, or a book.

Corpus:

- A collection of documents, usually in the same language and/or domain.

Text Preprocessing

Tokenization

Word Tokenization

Splitting text into individual words (or “tokens”).

Sentence Tokenization

Splitting text into individual sentences.

```
import re

s = "I love NLP!"
tokens = re.split(r"\s", s)
print(tokens)

s = "I love NLP! It's fascinating."
sentences = re.split(r"[.!?]", s)
print(sentences)
```

```
{'John': 1, 'likes': 2, 'to': 1, 'watch': 1,
'movies.': 1, 'Mary': 1, 'movies': 1, 'too.': 1}
['I', 'love', 'NLP!']
['I love NLP', " It's fascinating", '']
```

Lowercasing and Punctuation Removal

Lowercasing

Converting all characters in the text to lowercase.

```
-NLP is fascinating
+nlp is fascinating
```

Punctuation Removal

Removing punctuation marks from the text.

```
-Hello, world!
+Hello world
```

Stopwords Removal

Common words that do not carry much meaning and can be removed.

- Examples: "the", "is", "and", "a".

```
-This is a sample sentence, showing off the stop words filtr+This sample sentence, showing stop words filtration.
```

Stemming and Lemmatization

Stemming

Reduces words to a simpler base or “root” form.

- Example: "running", "runner", "ran" → "run"

```
-the bats saw the cats with stripes hanging upside down by t+the bat saw the cat with stripes hang upsid down by their fee
```

Lemmatization

Reduces words to their dictionary form (lemma) using vocabulary and morphological analysis.

- Example: "better" → "good" (using a dictionary that knows “better” is a form of “good”)

```
-the bats saw the cats with stripes hanging upside down by t+the bat see the cat with stripe hang upside down by their foo
```

Step 1: To Lower

Python has a built-in method to convert strings to lowercase.

```
# Original sentence
sentence = "The men will be running Wednesday if the weather gets better."
sentence = sentence.lower()
```

```
-The men will be running Wednesday if the weather gets bette+the men will be running wednesday if the weather gets better.
```

Step 2: Punctuation Removal

SpaCy can be used to remove punctuation. We load the English model and iterate through the tokens, removing punctuation marks.

```
import spacy

nlp = spacy.load("en_core_web_lg")  # Load spaCy model

doc = nlp(sentence)  # Initialize as a spaCy object (list of tokens)
words = []
for token in doc:
    if not token.is_punct:
        words.append(token.text)
sentence = ' '.join(words)
```

```
-the men will be running wednesday if the weather gets bette+the men will be running wednesday if the weather gets better
```

Step 3: Stopword Removal

Again, using spaCy to remove stopwords. We iterate through the tokens and filter out common words like "the" or "is."

```
doc = nlp(sentence)
words = []
for token in doc:
    if not token.is_stop:
        words.append(token.text)
sentence = ' '.join(words)
```

```
-the men will be running wednesday if the weather gets bette+men running wednesday weather gets better
```

Step 4: Stemming

NLTK provides a stemmer that can handle stemming:

```
from nltk.stem import PorterStemmer

doc = sentence.split()
stemmer = PorterStemmer()
words = []
for token in doc:
    words.append(stemmer.stem(token))
sentence = ' '.join(words)
```

```
-men running wednesday weather gets better
+men run wednesday weather get better
```

Step 5: Lemmatization

Lemmatization can also be done using spaCy. We iterate through the tokens and replace them with their lemma.

```
doc = nlp(sentence)
words = []
for token in doc:
    words.append(token.lemma_)
sentence = ' '.join(words)
```

```
-men run wednesday weather get better
+man run wednesday weather get well
```

*(Often, you choose to do either stemming **or** lemmatization, not both.)*

All Together Now

```python
import spacy
nlp = spacy.load("en_core_web_sm")  # Load small English model

sentence = "The men will be running Wednesday if the weather gets better."

preprocessed_tokens = []
doc = nlp(sentence)
for token in doc:
    # Filter out punctuation, stopwords, and spaces
    if not token.is_punct and not token.is_stop and not token.is_space:
        text = token.lemma_   # Lemmatized
        text = text.lower()   # Lowercase
        preprocessed_tokens.append(text)

print(sentence)
print(" ".join(preprocessed_tokens))
```

```
-The men will be running Wednesday if the weather gets bette+man run wednesday weather get well
```

A More Practical Preprocessor

```
import re

# Hardcoded stop words
STOP_WORDS = {"a", "an", "and", "are", "as", "at", "be", "by", "for", "from", "has", "he", "in", "is", "it", "its", "of", "
```

```
def preprocess(text):
    """Convert text to a list of lowercase words, removing stop words"""
    # Split on punctuation and whitespace
    tokens = re.split(r"[,\.\!\?\s]+", text)

    # Keep only non-empty lower-case tokens that aren't stop words
    processed_tokens = []
    for token in tokens:
        token = token.lower() # Lowercase the token
        if token and token not in STOP_WORDS:
            processed_tokens.append(token)

    return processed_tokens

# Test it
test_text = "Hello! How are you doing today?"
print(preprocess(test_text))  # Should print: ['hello', 'how', 'doing', 'today']
```

```
['hello', 'how', 'doing', 'today']
```

TF-IDF (Term Frequency–Inverse Document Frequency)

Reflects how important a word is to a document within a corpus.

Intuition

The word **"whale"** appears far more often in *Moby Dick* than in most books, so it's quite "unique" there. A common word like **"man"** might appear often, but that doesn't reveal much about the book's main topic. TF-IDF helps highlight words that are uniquely important to a document.

the most venerable of the leviathans, being the one first regularly hunted by man. It yields the article commonly known as whalebone or baleen; and the oil specially known as "whale oil," an inferior article in commerce. Among the fishermen, he is indiscriminately designated by all the following titles: The Whale; the Greenland Whale; the Black Whale; the Great Whale; the True Whale; the Right Whale. There is a deal of obscurity concerning the identity of the species thus multitudinously baptised. What then is the whale, which I include in the second species of my Folios? It is the Great Mysticetus of the English naturalists; the Greenland Whale of the English whalemen; the Baliene Ordinaire of the French whalemen; the Growlands Walfish of the Swedes. It is the whale which for more than two centuries past has been hunted by the Dutch and English in the Arctic seas; it is the whale which the American fishermen have long pursued in the Indian ocean, on the Brazil Banks, on the Nor' West Coast, and various other parts of the world, designated by them Right Whale Cruising Grounds.

the most venerable of the leviathans, being the one first regularly hunted by man. It yields the article commonly known as whalebone or baleen; and the oil specially known as "whale oil," an inferior article in commerce. Among the fishermen, he is indiscriminately designated by all the following titles: The Whale; the Greenland Whale; the Black Whale; the Great Whale; the True Whale; the Right Whale. There is a deal of obscurity concerning the identity of the species thus multitudinously baptised. What then is the whale, which I include in the second species of my Folios? It is the Great Mysticetus of the English naturalists; the Greenland Whale of the English whalemen; the Baliene Ordinaire of the French whalemen; the Growlands Walfish of the Swedes. It is the whale which for more than two centuries past has been hunted by the Dutch and English in the Arctic seas; it is the whale which the American fishermen have long pursued in the Indian ocean, on the Brazil Banks, on the Nor' West Coast, and various other parts of the world, designated by them Right Whale Cruising Grounds.

TF-IDF Calculation

1. **Term Frequency (TF)**: How often a word appears in a specific document (normalized by total words in that doc):

$$\mathrm{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

   where $f_{t,d}$ is the count of term t in document d, and $\sum_{t' \in d} f_{t',d}$ is the total word count in d.

2. **Inverse Document Frequency (IDF)**: Measures how rare a word is across all documents in the corpus:

$$\mathrm{idf}(t, D) = \log\left(\frac{N}{|\{d : d \in D \text{ and } t \in d\}|}\right)$$

   where N is the total number of documents.

3. **TF-IDF**: The product of the two:

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

Step 1: Define Your Corpus and Pick a Target Document

```
corpus = [
    "John likes to watch movies.",
    "Mary likes movies too.",
    "John also likes to watch football games."  # <-- Target Document
]
tgt_doc = corpus[2]
```

A "corpus" is just a collection of documents. For example:

- “Moby Dick” = one document
- “All English Books” = one corpus

Step 2: Term Frequency (TF)

Calculate the frequency of each term in the *target* document.

```
def calculate_term_frequency(text):
    term_freq = {}
    for term in preprocess(text):
        if term in term_freq:
            term_freq[term] += 1
        else:
            term_freq[term] = 0
    return term_freq
```

Step 3: Document Frequency (DF)

Track how many documents (in the entire corpus) contain each word.

```
def calculate_document_frequency(corpus, target_terms):
    doc_freq = {}

    # First, preprocess all of the documents in the corpus
    preprocessed = []
    for doc in corpus:
        preprocessed.append(preprocess(doc))

    # Next, for each term in the target terms, count how many documents contain it
    for term in target_terms:
        for doc in preprocessed:
            if term in doc:
                if term in doc_freq:
                    doc_freq[term] += 1
                else:
                    doc_freq[term] = 1
    return doc_freq
```

Step 4: Calculate TF-IDF

```
def calculate_tfidf(term_freq, doc_freq, total_docs):
    tfidf = {}
    for word in term_freq:
        idf = math.log(total_docs / doc_freq[word])
        tfidf[word] = term_freq[word] * idf
    return tfidf
```

IDF

Compute how rare or common each word is across the corpus. (Don't forget to import `math` for `math.log`!)

TF-IDF

Multiply the term frequency by the inverse document frequency to get the TF-IDF score for each word in the target document.

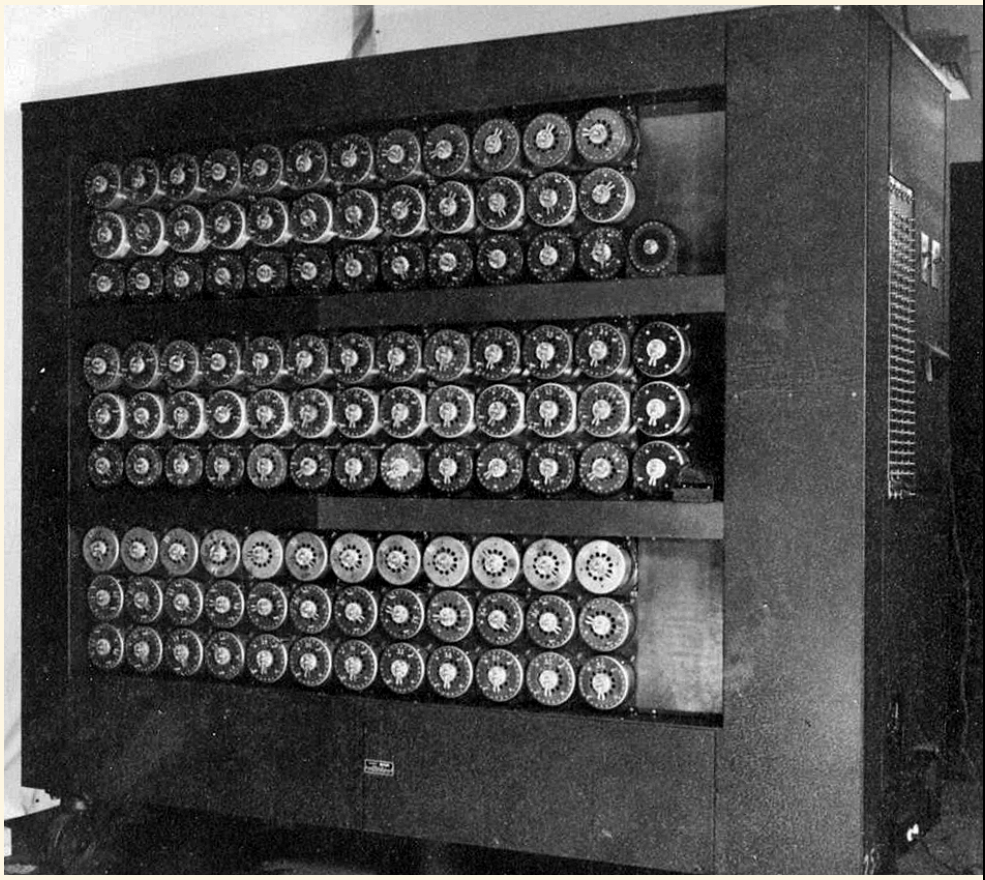
Exercise: TF-IDF on Lyrics

bigd103.link/tfidf-lyrics

Basic Text Representation

Problem Statement

- Machines need numbers, not raw text.
- Machines work best with fixed-sized vectors (lists) of numbers.
- **Goal**: Convert text data into useful numerical features.

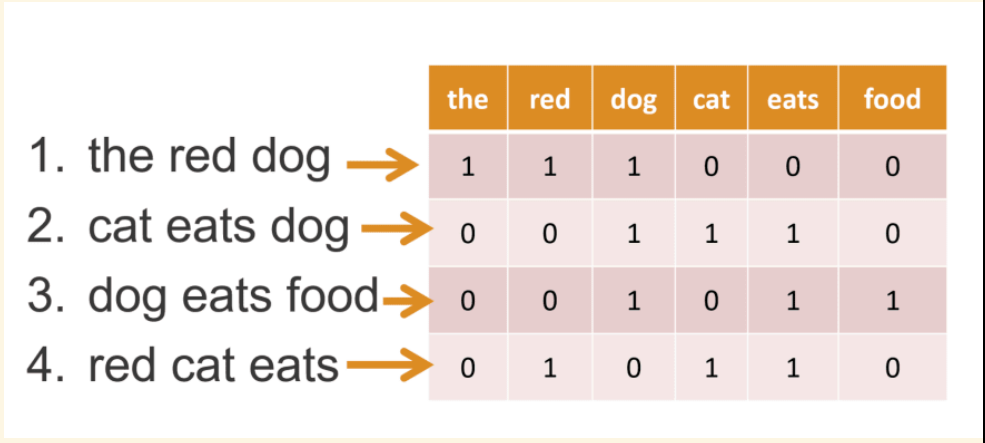


Bag of Words (BoW)

Represents text by counting the occurrences of each word in a document, ignoring grammar and word order.

Algorithm:

1. Create a “vocabulary” of unique words from the corpus.
2. For each document, represent it as a vector of word counts.



Bag of Words Example

```
sentence = "John likes to watch movies. Mary likes movies too."
bow = {}
for word in sentence.split():
    bow[word] = bow.get(word, 0) + 1
print(bow)
```

```
<exec>:2: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466


{'John': 1, 'likes': 2, 'to': 1, 'watch': 1, 'movies.': 1, 'Mary': 1, 'movies': 1, 'too.': 1}
```

Bag of Words

Advantages

1. **Simplicity** – Easy to implement and understand.
2. **Interpretability** – The frequency-based representation is intuitive and straightforward.
3. **Small Datasets Friendly** – Effective for simple text classification tasks with limited data.

Disadvantages

1. **Ignores Word Order** – Loses context and syntax information.
2. **High Dimensionality** – Large vocabularies mean big, sparse vectors.
3. **Vocabulary Sensitivity** – Requires careful feature selection (e.g., removing stopwords) to reduce noise.

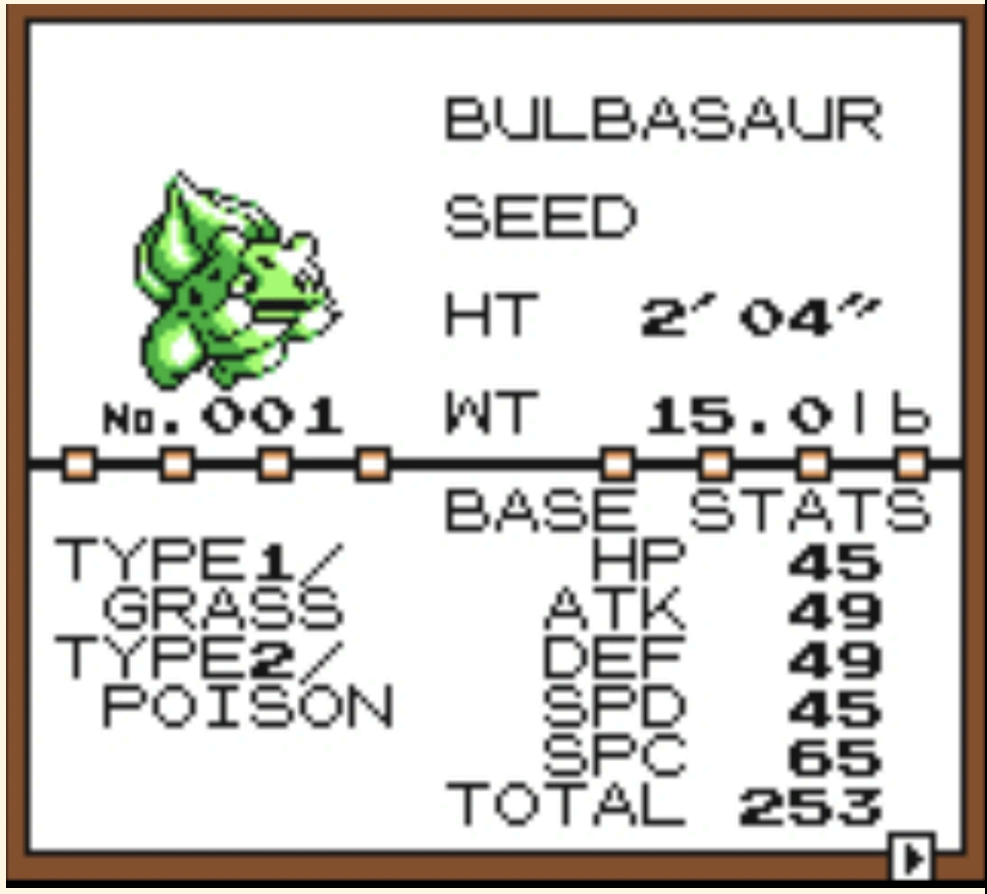
Word Embeddings

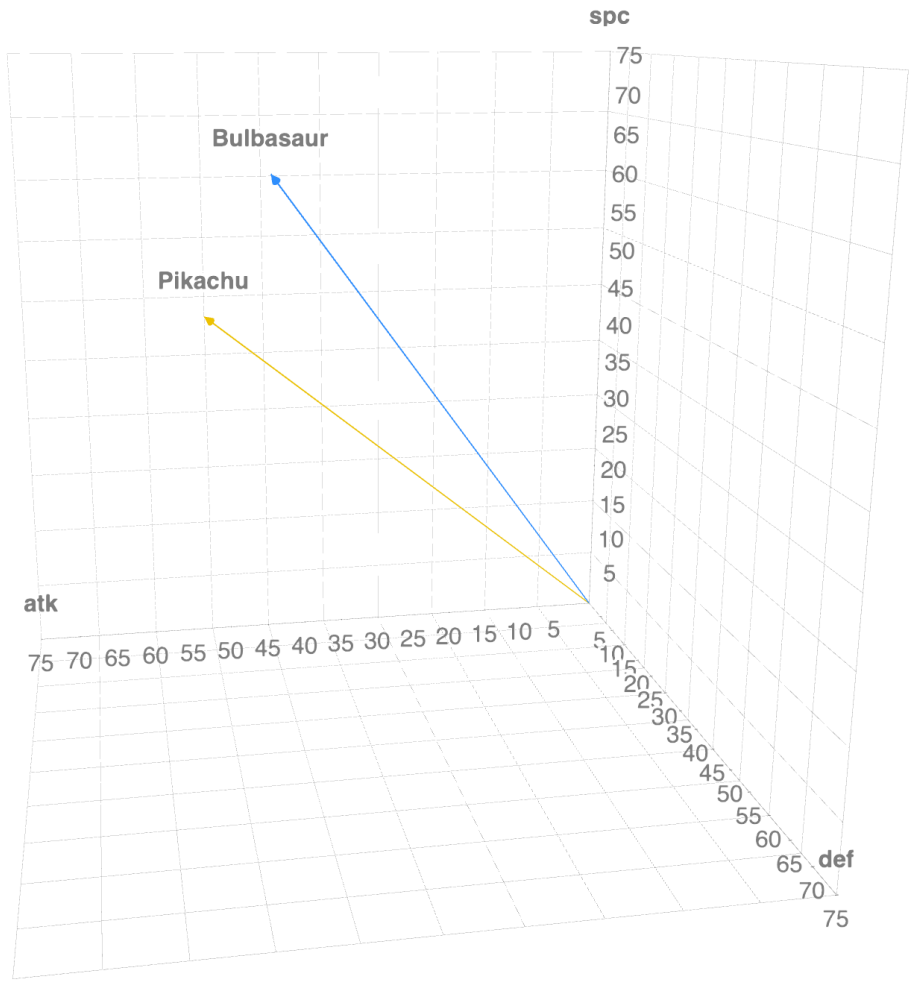
First, What is an Embedding?

Let's consider the humble Bulbasaur.

```
df.set_index("Name", inplace=True)
df[["HP", "Attack", "Defense", "Sp. Atk", "Sp. Def",
df.loc["Bulbasaur"]
```

```
HP         45
Attack     49
Defense    49
Sp. Atk    65
Sp. Def    65
Speed      45
Name: Bulbasaur, dtype: int64
```

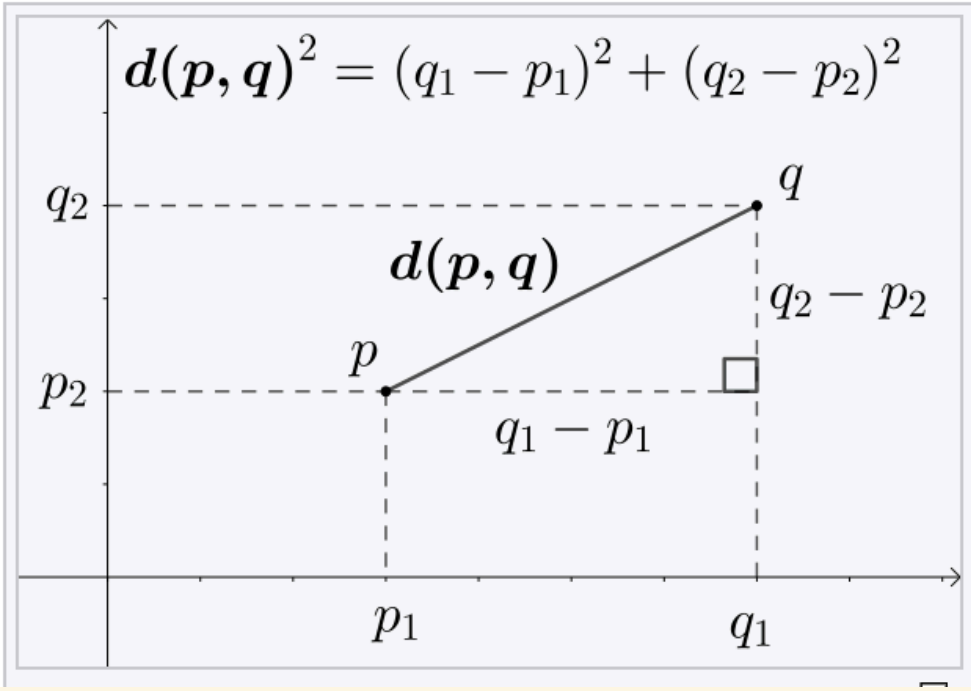



spc
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
Bulbasaur
Pikachu
atk
75 70 65 60 55 50 45 40 35 30 25 20 15 10 5
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
def

Distance Between Bulbasaur and Pikachu

Each Pokemon can be thought of as a vector in a 6-dimensional space. And therefore, we can calculate the distance between them the same way we would calculate the distance between two points in 2-dimensional space.

```
p = df.loc["Bulbasaur"].values
q = df.loc["Pikachu"].values

np.linalg.norm(p - q)
```

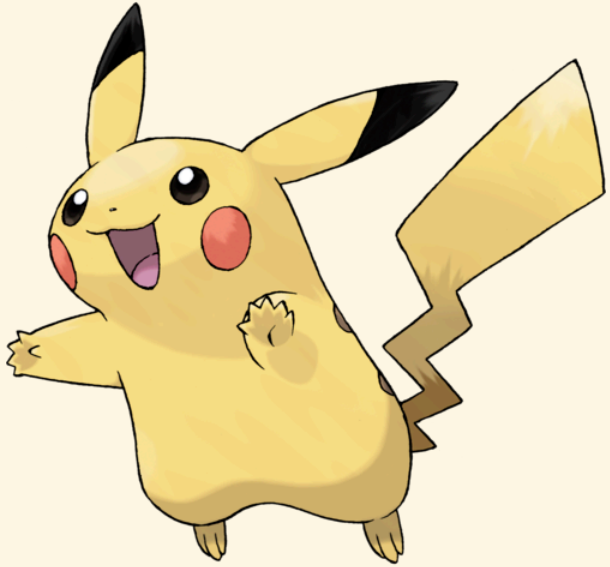


```
df.loc[['Bulbasaur', 'Vanillite']]
```

Name	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed
Bulbasaur	45	49	49	65	65	45
Vanillite	36	50	50	65	60	44

```
df.loc[['Pikachu', 'Diglett']]
```

```
         HP  Attack  Defense  Sp. Atk  Sp. Def  Speed
Name
Pikachu  35      55       40       50       50     90
Diglett  10      55       25       35       45     95
```

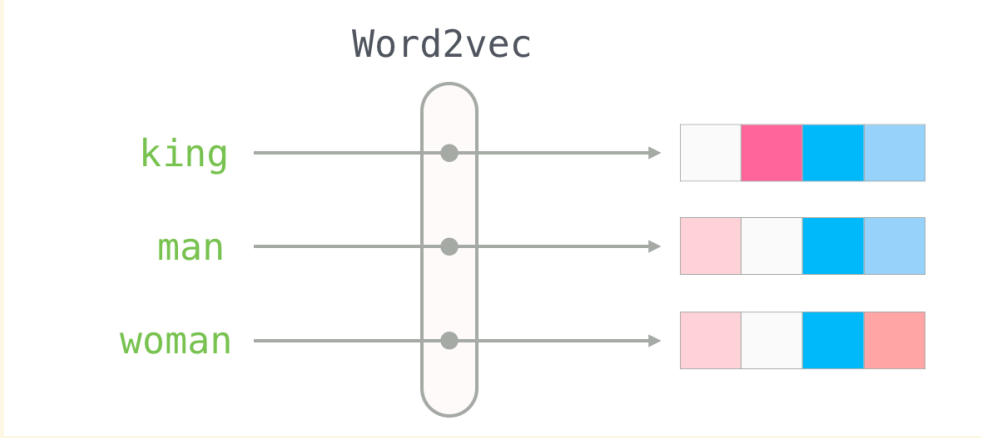


```
df.loc[['Nidoqueen', 'Poliwrath']]
```

Name	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed
Nidoqueen	90	92	87	75	85	76
Poliwrath	90	95	95	70	90	70

Word Embeddings

- Word embeddings are dense, low-dimensional vectors.
- They capture semantic relationships between words.
- Words with similar meanings have embeddings that end up being close together.



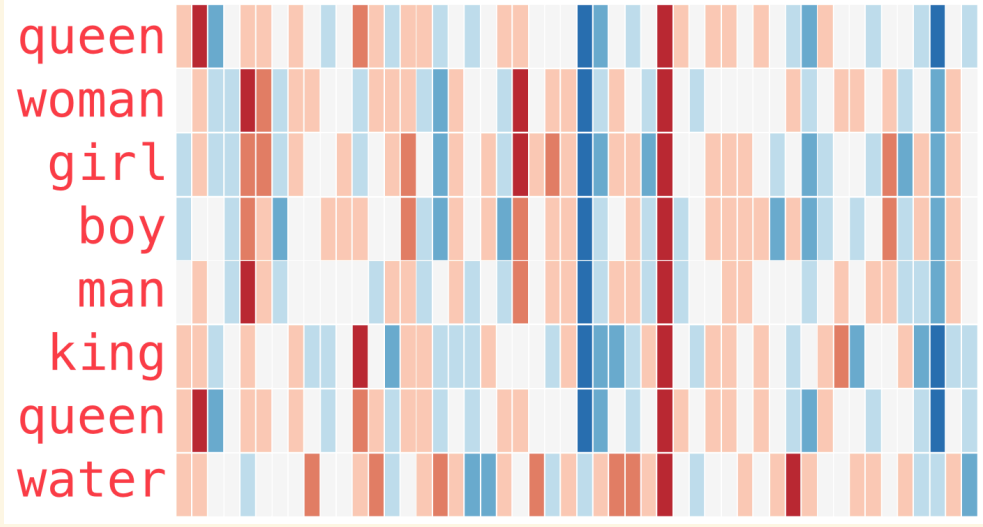
Comparing Word Vectors

Semantic Similarity

Words that appear in similar contexts (e.g., “king” and “queen” or “dog” and “puppy”) tend to have similar vector representations.

Simple Vector Arithmetic (Analogies)

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$



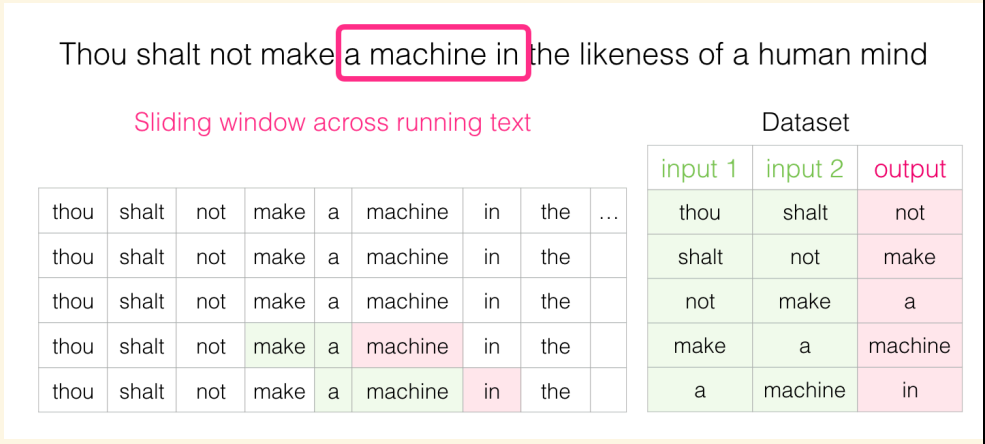
Word2Vec

- Word2vec was developed by Tomáš Mikolov and colleagues at Google and published in 2013. -Vectors capture information about the meaning of the word based on the surrounding words.
- The word2vec algorithm estimates these representations by modeling text in a large corpus.
- Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence.

Word Embeddings with SpaCy

```
import spacy
import numpy as np

nlp = spacy.load("en_core_web_lg")  # Large English
cat = nlp("cat")
dog = nlp("dog")
ham = nlp("ham")

print(f"Distance from 'cat' to 'dog' is {np.linalg.n
print(f"Distance from 'dog' to 'ham' is {np.linalg.n
```

```
Distance from 'cat' to 'dog' is 42.8679084777832
Distance from 'dog' to 'ham' is 77.25950622558594
```



Some More SpaCy Stuff

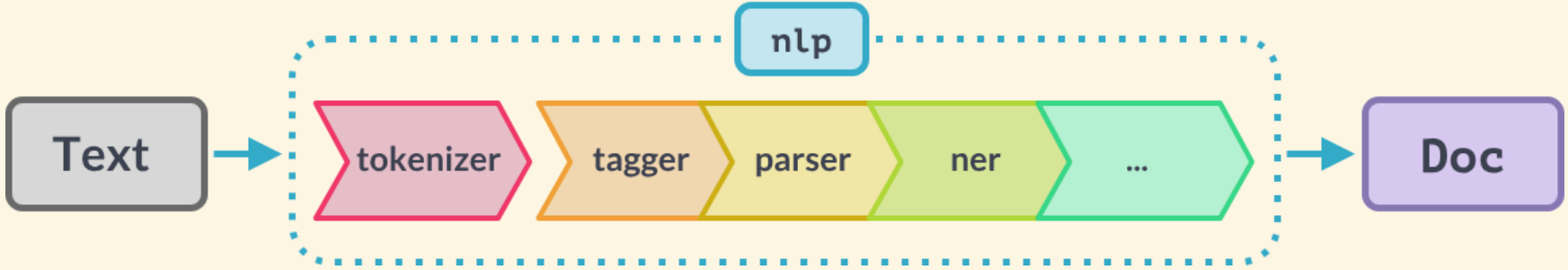
Faster SpaCy

One technique to speed up spaCy is to parallelize the processing of the text. This can be done using the `nlp.pipe` method.

```
import spacy
from tqdm.notebook import tqdm

nlp = spacy.load("en_core_web_sm")

df["doc"] = list(tqdm(nlp.pipe(df['text'], n_process=4), total=len(df)))

df["preprocessed_text"] = df["doc"].apply(preprocess)
```

SpaCy Processing Pipelines

When you call `nlp` on a text, spaCy first tokenizes the text to produce a Doc object. The Doc is then processed in several different steps – this is also referred to as the processing pipeline



SpaCy supports optional libraries that extend this pipeline with new features such as detecting language.

```
import spacy
import spacy_fastlang
nlp = spacy.load("en_core_web_lg")
nlp.add_pipe("language_detector")
df["doc"] = list(nlp.pipe(df['text'], n_process=4))
df["language"] = df["doc"].apply(lambda doc: doc._.language)
```

SpaCy Parts of Speech

In addition to lemmatization and stop-word removal, spaCy can also be used to get the part of speech of a word. This lets us extract things like nouns from text.

```
def parse_nouns(doc):
    """Returns a string of just the proper nouns lowercased."""
    nouns = []
    for token in doc:
        if token.pos_ == "NOUN" or token.pos_ == "PROPN":
            nouns.append(token.text.lower())
    return " ".join(nouns)

df["nouns"] = df["doc"].apply(parse_nouns)
```

Sentiment Analysis

Sentiment Analysis

A technique in NLP for identifying the emotional tone behind words. It determines if text expresses a positive, negative, or neutral attitude.

Goals:

- Quickly interpret the emotional content of large amounts of text.
- Provide actionable insights (e.g., analyzing customer reviews).

Common Use Cases:

- **Customer Feedback**: Product reviews, support tickets.
- **Brand Monitoring**: Gauging social media sentiment.
- **Finance & Markets**: Predicting trends from news or investor sentiment.
- **Politics & Public Opinion**: Evaluating reactions to policies, speeches, etc.

Simple Lexicon-Based Sentiment Analysis

- Maintain a list (“lexicon”) of words, each labeled with a sentiment score (positive, negative, neutral).
- Count or sum the scores of the words in the text.
- Adjust for negations (e.g., “not good”) or intensifiers (e.g., “very happy”).

Example: “The movie was amazing and inspiring!”

- "amazing" → +3, "inspiring" → +2
- Total Score = +5 (Positive)

Example: “The service was not good.”

- "good" → +2, but "not" can flip the meaning
- Adjusted Score = -2 (Negative)

Simple Lexicon-Based Sentiment with SpaCy

SpaCy's default English models do not include a built-in sentiment classifier. However, we can roll our own simple version by maintaining a small dictionary ("lexicon") of words pre-labeled with sentiment scores (e.g., positive and negative words) and then counting their occurrences in a piece of text.

How It Works:

1. Choose or create a lexicon of positive and negative words.
2. Convert input text to a spaCy Doc object.
3. For each token, check if it's in our lexicon of positive or negative words.
4. Sum up all token scores to get an approximate sentiment.

(This is a simplistic approach—real-world systems typically use more advanced machine-learning or large language model techniques.)

Example Lexicon

Below is a tiny example lexicon that assigns +1 to positive words and -1 to negative words. Words not in the lexicon will contribute 0 to the overall sentiment score.

```
POSITIVE_WORDS = {
    "amazing", "awesome", "good", "great", "wonderful", "love", "happy", "fantastic"
}

NEGATIVE_WORDS = {
    "bad", "terrible", "awful", "sad", "unhappy", "hate", "poor", "worse"
}
```

(In practice, your lexicon might have hundreds or thousands of words, possibly with different numeric scores.)

Lexicon-Based Sentiment Analysis in SpaCy

```
POSITIVE_WORDS = {...}
NEGATIVE_WORDS = {...}

def get_sentiment_score(text):
    """
    Returns a simple sentiment score by counting words
    in the text that appear in our POSITIVE_WORDS or NEGATIVE_WORDS sets.
    """
    doc = nlp(text.lower())  # spaCy Doc object (lowercased to match our sets)
    score = 0
    for token in doc:
        lemma = token.lemma_   # get lemma so "loved" -> "love"
        if lemma in POSITIVE_WORDS:
            score += 1
        elif lemma in NEGATIVE_WORDS:
            score -= 1
    return score

sentence = "I am very unhappy with this product, but I love the packaging!"
score = get_sentiment_score(sentence)
```

Sentiment Analysis with TextBlob

```
import spacy
from textblob import TextBlob

nlp = spacy.load("en_core_web_sm")
sentence = "I am very unhappy with this product."
doc = nlp(sentence)

blob = TextBlob(doc.text)

print(f"Sentiment Polarity: {blob.sentiment.polarity}")
print(f"Sentiment Subjectivity: {blob.sentiment.subjectivity}")
print(f"Assessments: {blob.sentiment_assessments.assessments}")
```

```
Sentiment Polarity: -0.78
Sentiment Subjectivity: 1.0
Assessments: ([['very', 'unhappy'], -0.78, 1.0, None)]
```

Exercise!

bigd103.link/sentiment-lyrics

TF-IDF Applied to Subsets

Subset TF-IDF: Finding What Makes Groups Unique

From Single Documents to Groups

- **TF-IDF**: What makes *one song* unique in the corpus?
- **Subset TF-IDF**: What makes *all songs by an artist* unique in the corpus?

Examples:

- What words define Taylor Swift's songwriting style?
- What terms are distinctive to hip-hop vs country music?
- What vocabulary is unique to 2020s music vs 1980s music?

Subset-TF-IDF Calculation

1. **Term Frequency (TF) for the Subset**: Combine term frequencies across all documents in the subset:

$$\mathrm{tf}(t, S) = \frac{\sum_{d \in S} f_{t,d}}{\sum_{d \in S} \sum_{t' \in d} f_{t',d}}$$

   where S is the target set of documents (e.g., all songs by a specific artist).

2. **Inverse Document Frequency (IDF)**: Compute IDF over all documents:

$$\mathrm{idf}(t, D) = \log\left(\frac{N}{|\{d \in D : t \in d\}|}\right)$$

   where N is the total number of documents (songs) in the corpus.

3. **Subset TF-IDF**: Multiply the aggregated TF by the global IDF:

$$\mathrm{tfidf}(t, S, D) = \mathrm{tf}(t, S) \cdot \mathrm{idf}(t, D)$$

   This score reflects how uniquely important a term is for that subset (artist) relative to the entire

Step 1: Define Your Subset

Instead of one target document, we now have a target *subset* of documents.

```
# Example: All songs by a specific artist
all_songs = [
    {"artist": "Taylor Swift", "lyrics": "shake it off shake it off"},
    {"artist": "Taylor Swift", "lyrics": "blank space write your name"},
    {"artist": "Dua Lipa", "lyrics": "new rules counting them"},
    {"artist": "Dua Lipa", "lyrics": "levitating moonlight"},
]

# Define our subset: all Taylor Swift songs
target_subset = [doc for doc in all_songs if doc["artist"] == "Taylor Swift"]
```

Step 2: Aggregate Term Frequency

Calculate term frequency across ALL documents in the subset.

```
def calculate_subset_term_frequency(subset_docs):
    subset_tf = {}
    # Count terms across ALL documents in the subset
    for doc in subset_docs:
        for term in preprocess(doc):
            if term in subset_tf:
                subset_tf[term] += 1
            else:
                subset_tf[term] = 1
    return subset_tf
```

Key difference: We're counting terms across multiple documents, not just one!

Step 3: Document Frequency (Same as TF-IDF)

Count how many documents in the ENTIRE corpus contain each term.

```
def calculate_document_frequency(corpus, target_terms):
    doc_freq = {}
    # This is exactly the same as regular TF-IDF!
    for term in target_terms:
        count = 0
        for doc in corpus:
            if term in preprocess(doc):
                count += 1
        doc_freq[term] = count
    return doc_freq
```

Note: We still use the full corpus, not just the subset!

Step 4: Calculate Subset TF-IDF

The formula is the same, but now TF represents the subset's term frequency.

```
def calculate_subset_tfidf(subset_tf, doc_freq, total_docs):
    subset_tfidf = {}
    for word in subset_tf:
        # IDF: How rare is this word in the full corpus?
        idf = math.log(total_docs / doc_freq[word])
        # Subset TF-IDF: How important is this word to the subset?
        subset_tfidf[word] = subset_tf[word] * idf
    return subset_tfidf
```

High scores = words that are:

1. Frequent in the subset (high TF)
2. Rare in the overall corpus (high IDF)

Example: Artist Analysis

```
# Get all Taylor Swift songs
taylor_songs = df[df["Artist"] == "Taylor Swift"]["Lyric"].tolist()

# Calculate subset TF (all her songs combined)
all_taylor_lyrics = " ".join(taylor_songs)
subset_tf = calculate_term_frequency(all_taylor_lyrics)

# Use the same corpus and doc_freq from before
subset_tfidf = calculate_subset_tfidf(subset_tf, doc_freq, len(corpus))

# Top words that define Taylor Swift's style
print("Words unique to Taylor Swift:")
for word, score in sorted(subset_tfidf.items(), key=lambda x: x[1], reverse=True)[:10]:
    print(f"  {word}: {score:.3f}")
```

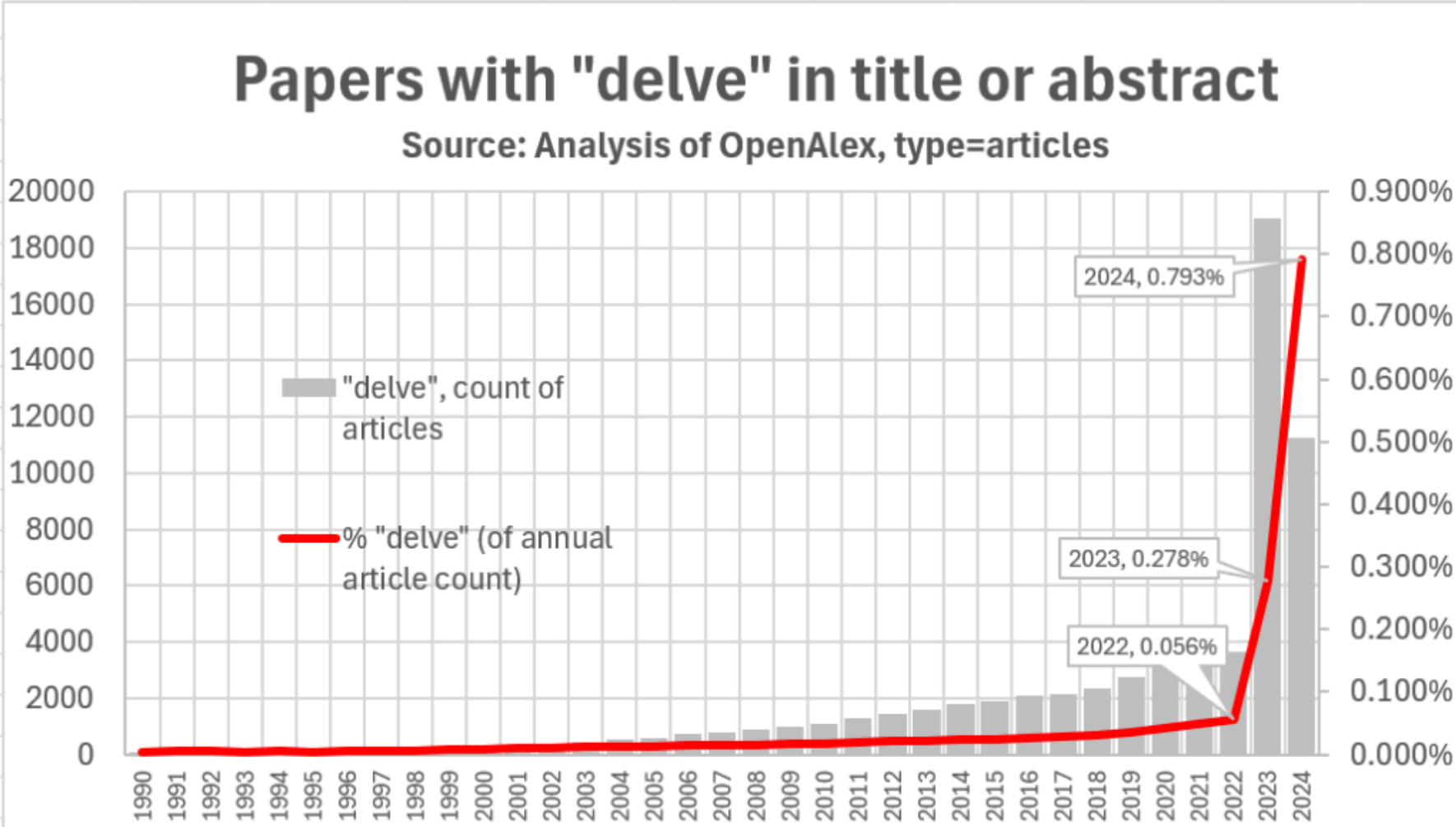
When and Why to Use Subset-TF-IDF

- **Use Case**:
  When you want to determine what differentiates a subgroup (e.g., all songs by one artist) from a broader collection of documents (e.g., songs by various artists).

- **Benefits**:

  - **Contextual Relevance**: Highlights themes and vocabulary specific to the artist's style.
  - **Insightful Comparisons**: Helps compare the language and topics of different artists or groups.
  - **Aggregated Trends**: Smooths out anomalies from a single document by considering a broader set of texts.

- **Example**:
  Instead of performing TF-IDF on one song, aggregating TF-IDF scores over all of Artist A's songs can reveal terms like **"moonlight"**, **"soul"**, or **"rhythm"** that capture the essence of their lyrical style, distinguishing them from other artists in the dataset.



Analysis using OpenAlex by Philip Shapira, March 31, 2024.

Exercise: Subset TF-IDF on NSF Grants

bigd103.link/nsf-grants