**Introduction to preprocessing for text**

**What we will learn**

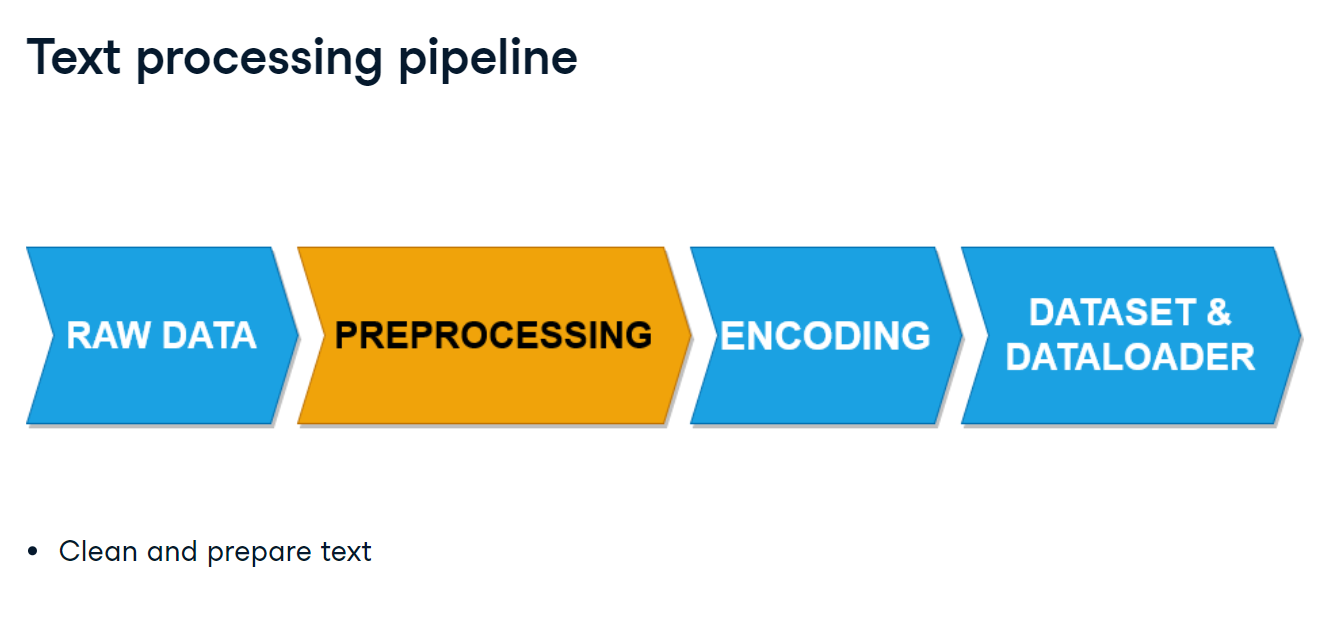
We will explore deep learning using PyTorch for text classification and generation. We'll cover encoding, deep learning models for text, and advanced topics around transformer architecture and protecting our models from attacks. These skills apply to real-world tasks, like sentiment analysis, text summarization, and machine translation.

Before we begin, you should already be familiar with developing deep learning models with PyTorch, including training and evaluation loops, and have familiarity with convolutional and recurrent neural networks.

**Text processing pipeline**

Welcome to the text processing pipeline! Our text analysis approach in PyTorch involves preprocessing, encoding, and Dataset and DataLoader. This video will focus on preprocessing. We will explore encoding and recap Dataset and Dataloader later in the chapter. Let's begin.

In preprocessing, we clean and prepare the text data for encoding.



**PyTorch and NLTK**

Preprocessing raw text data utilizes natural language processing techniques. We'll use PyTorch and NLTK, the Natural Language Toolkit, which provides a range of techniques to transform raw text into processed text.

**Preprocessing techniques**

We will discuss tokenization, stop word removal, stemming, and rare word removal.

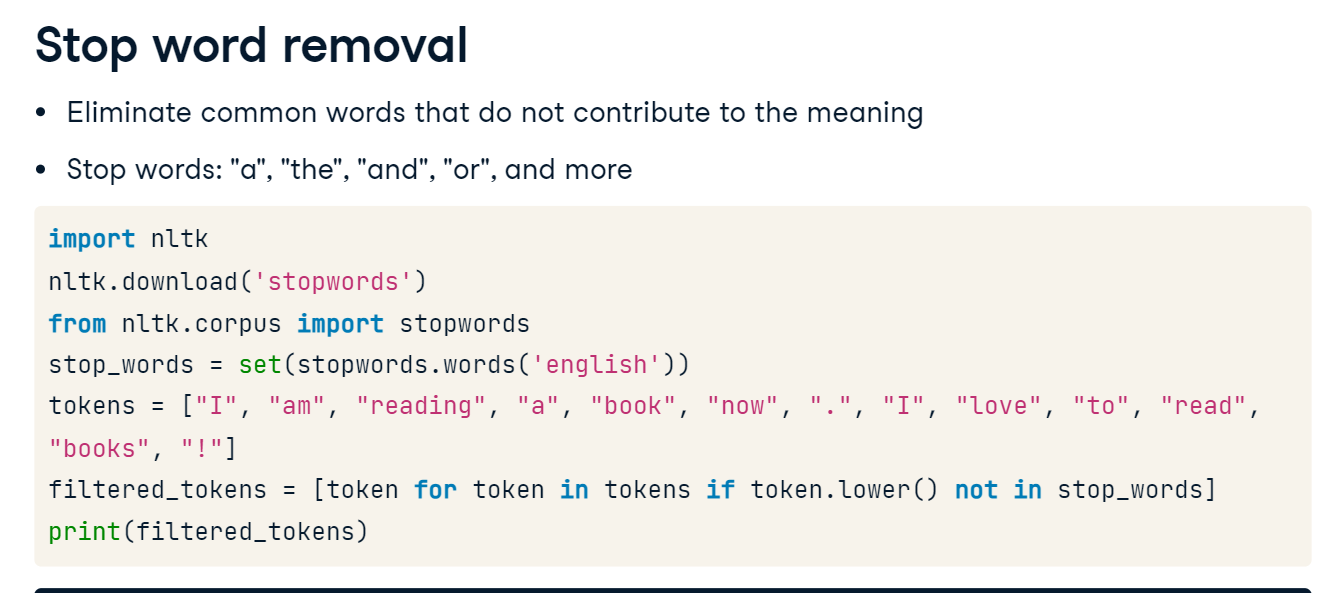
**Tokenization**

The first step in text preprocessing is tokenization. This is where we extract tokens from text. A token could be a full word, part of a word, or a punctuation. We'll use the PyTorch get\_tokenizer function imported from torchtext-dot-data-dot-utils. The basic\_english tokenizer supports the English language. We input the sentence: "I am reading a book now. I love to read books!". By applying tokenization, our output becomes a list of tokens.



**Stop word removal**

Next is stopword removal, where NLTK is more suited. Here, we eliminate stopwords or commonly occurring words such as a, the, and, or, and others that don't contribute to the meaning of a text, allowing the model to focus on the words with meaning. We download the stopwords collection of words, also known as corpus, from nltk using nltk-dot-download and import the stopwords package. We create a set of stopwords with no duplicates using stopwords-dot-words. We use English to process English text, but other options are available. With list comprehension, we iterate through the tokens we previously created and filter out any stopwords. Note the use of the lower method; this helps us capture all instances of stopwords regardless of capitalization. Finally, we print the filtered tokens.



**Stemming**

Stemming reduces words or tokens to their base or root form for simplified analysis. For example, "running," "runs," and "ran" would all be converted to "run" using stemming. We use the NLTK library's PorterStemmer package to perform stemming on a set of words or tokens. We initialize the PorterStemmer. Its input will be a list of tokenized words with stopwords removed. We iterate through this list using stemmer-dot-stem to stem each token. In the output, reading becomes read, and books becomes book.



**Rare word removal**

Lastly, we can remove rare words that occur infrequently and may not provide value for our text analysis. We calculate the word frequencies using the FreqDist function from the nltk-dot-probability module and define the tokens input. We then define a threshold value of two to determine the rare words. We filter out the rare words by keeping only tokens whose frequency exceeds the threshold. Then, we print the result.



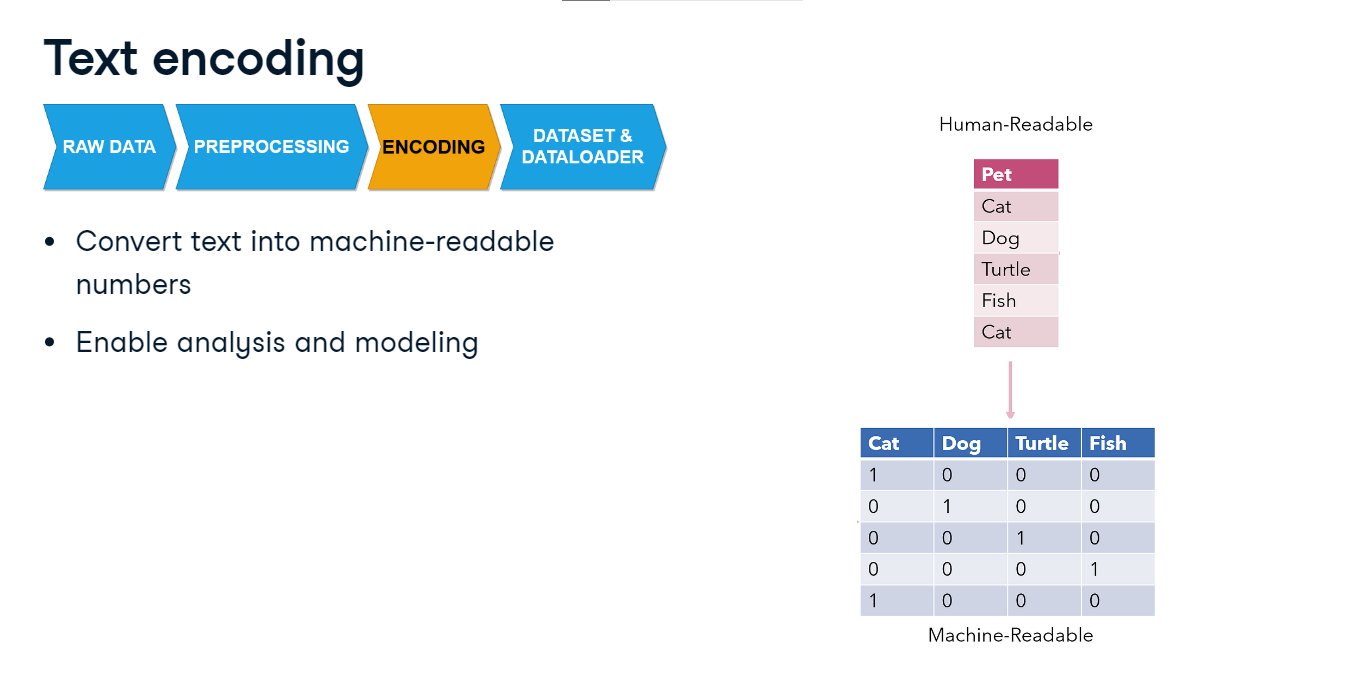
**Preprocessing techniques**

The techniques we have covered help refine our text data by reducing the number of features and creating cleaner, more representative datasets. We have only covered a few techniques here. Many more exist but are out of scope for this course. We encourage you to explore these further.

**Encoding text data**

**Text encoding**

Encoding happens after processing the data. Using PyTorch, we can convert text into machine-readable numbers for analysis and modeling. As seen in the image, each value in the red table is encoded in the blue table.

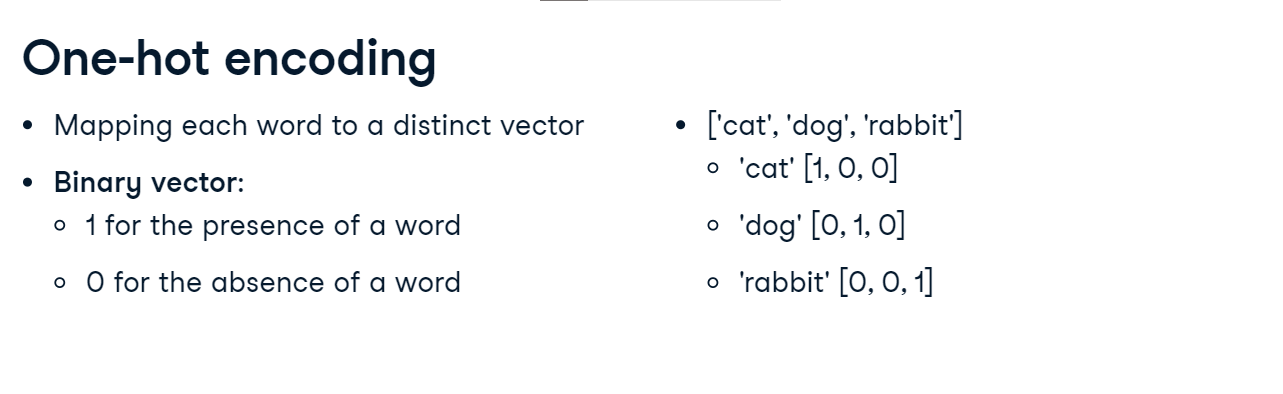


**Encoding techniques**

We will discuss three encoding methods: One-hot encoding transforms words into unique numerical representations, Bag-of-Words captures word frequency disregarding order, and TF-IDF balances the uniqueness and importance of words in a document. Additionally, embedding converts words into vectors representing semantic meanings. We will review embeddings in the next chapter.

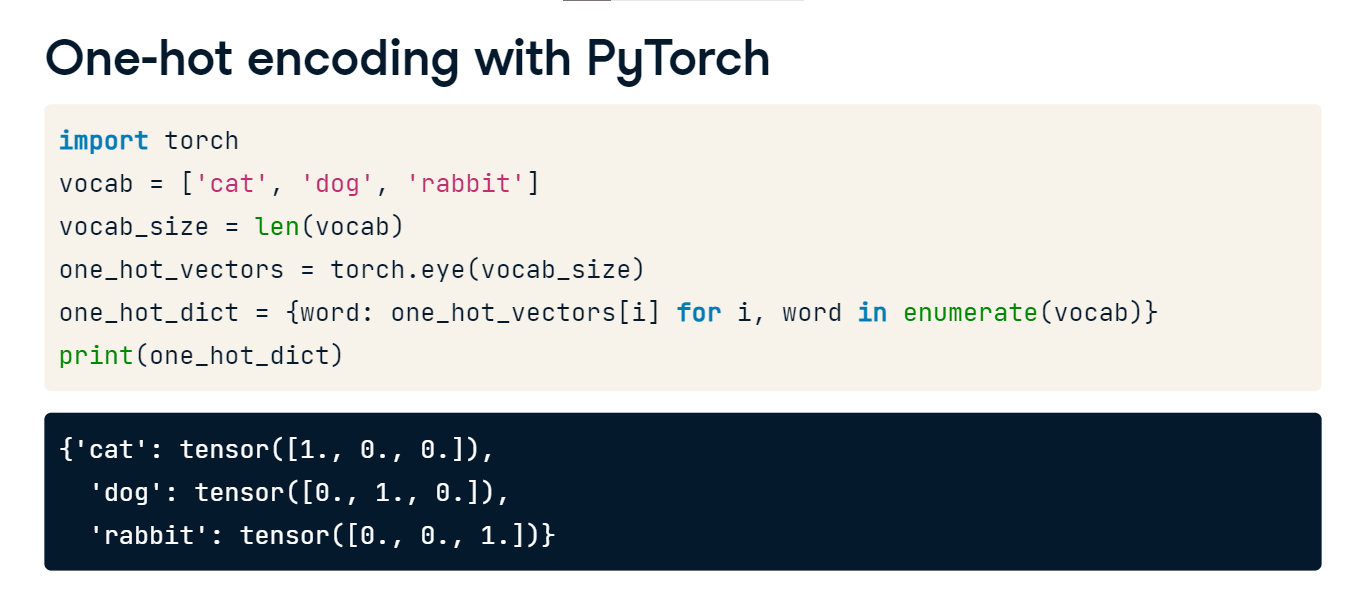
**One-hot encoding**

With one-hot encoding, each word maps onto a distinct one-hot binary vector within the encoding space where one represents the presence of a word and zero the absence. For instance, in a vocabulary consisting of cat, dog, and rabbit, the one-hot vector for 'cat' could be [1, 0, 0], [0, 1, 0] for 'dog' and [0, 0, 1] for 'rabbit'.



**One-hot encoding with PyTorch**

We have a vocab list that contains input tokens. For sentence input, we tokenize to create a list of tokens. We first determine the vocab list length. Using torch, we utilize the torch-dot-eye function to generate one-hot vectors for the length of our list. We create a dictionary called one\_hot\_dict where each word is mapped to its corresponding vector from one\_hot\_vectors. This allows us to easily access the vector representation of any word in our vocabulary.



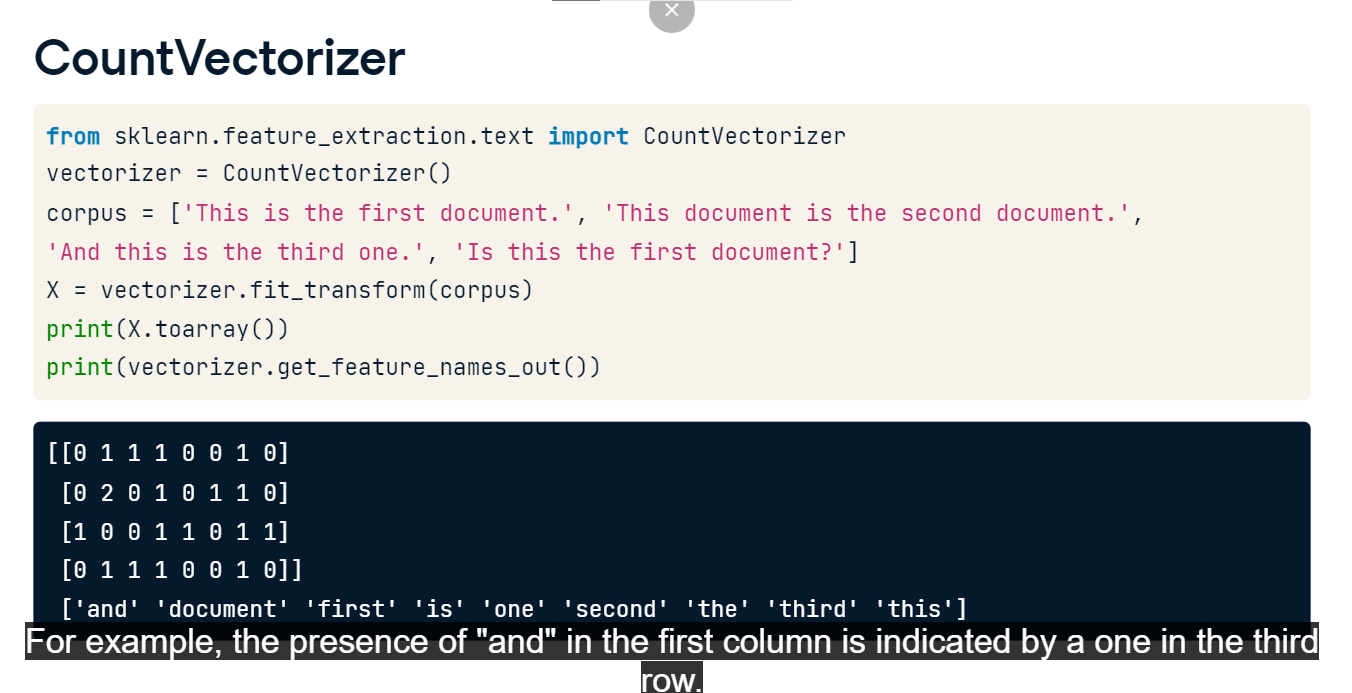
**Bag-of-words**

Alternatively, we could improve our models by adding more meaning with bag-of-words, which treats a document as an unordered collection of words, emphasizing word frequency over order. For instance, the sentence 'The cat sat on the mat' is converted into a dictionary. In our case, "the" is the only word that appears twice.



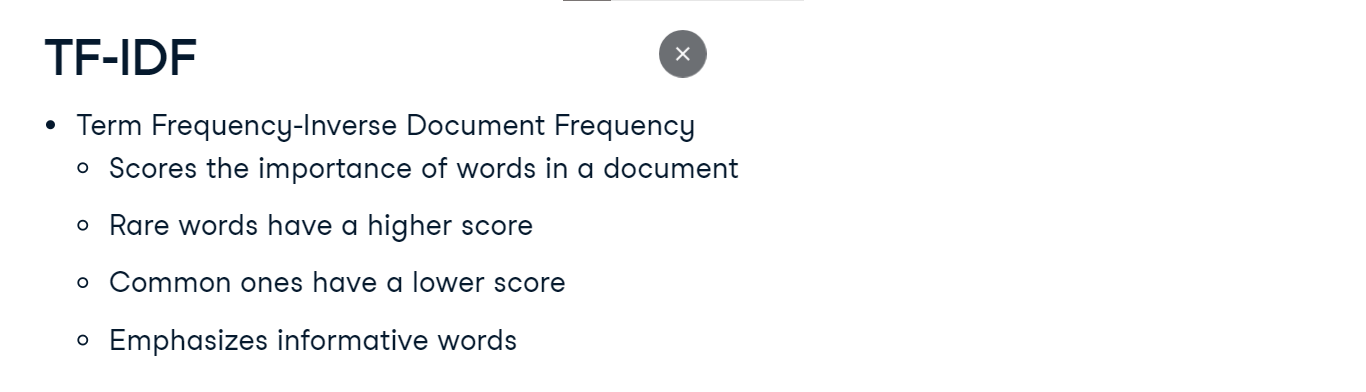
**CountVectorizer**

In some cases, like this one, sklearn streamlines Bag-of-Words implementation. We import CountVectorizer from sklearn-dot-feature\_extraction-dot-text. We instantiate a CountVectorizer object. We define our corpus, a collection of text documents represented here as a list of sentences. This can also be a tokenized list. We fit our vectorizer to the corpus and transform it into a numerical format using fit\_transform. This produces our Bag-of-Words representation, which we store in X and print using the toarray function. We can visualize the words by extracting the feature names from the vectorizer with dot-get\_feature\_names\_out. The output is a term frequency matrix, where each row corresponds to a document and each column corresponds to a word. For example, the presence of "and" in the first column is indicated by a one in the third row.



**TF-IDF**

The last technique we will cover is TF-IDF or Term Frequency-Inverse Document Frequency. It assesses word importance by considering word frequency across all documents, assigning higher scores to rare words and lower scores to common ones. TF-IDF emphasizes informative words in our text data, unlike bag-of-words, which treats all words equally.



**TfidfVectorizer**

To use TF-IDF we import TfidfVectorizer from sklearn. We instantiate a TfidfVectorizer object using the same corpus as before and fit it like we did for CountVectorizer. This transforms the data into TF-IDF vectors. TF-IDF can also accept a tokenized list. The toarray function yields a matrix of TF-IDF scores. We print the feature names. Every row in the matrix represents a document from the corpus. The feature names list displays the most significant words across all documents, and each word represents a column of the matrix.

For instance, the importance of the word first is highest in the first sentence with a score of zero-point-six-eight.



**Encoding techniques**

Encoding allows models to understand and process text. Ideally, we choose one technique for encoding to avoid redundant computations. As with processing, other encoding techniques exist but are beyond this course's scope. We will cover embeddings in the next chapter.