**Overview of Text Classification**

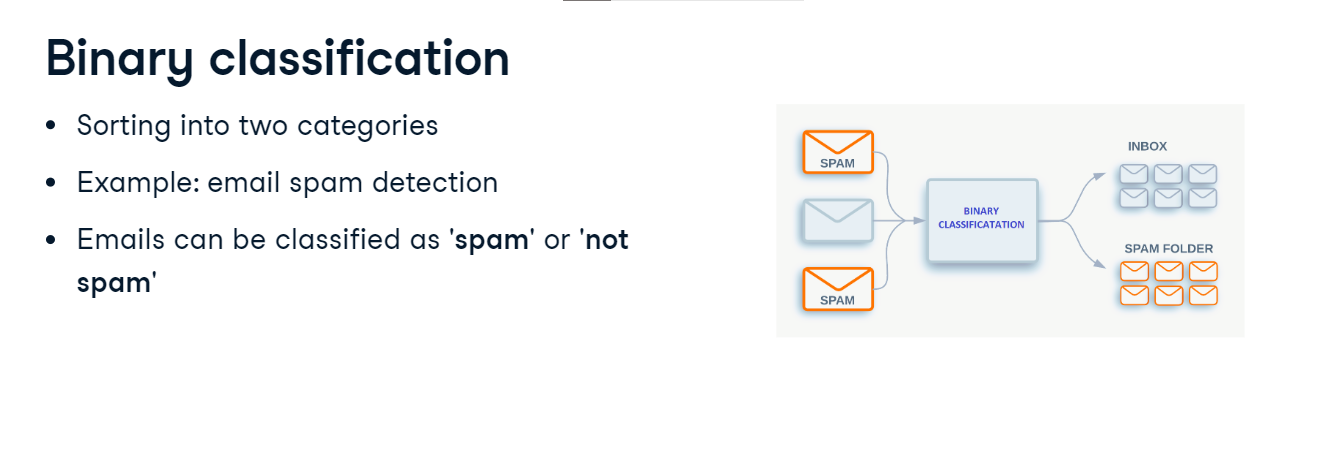
**Text classification defined**

Text classification assigns labels to text, giving meaning to words and sentences. It helps in organizing and giving structure to unstructured data and is crucial in various applications, such as analyzing customer sentiment in reviews, detecting spam emails, or tagging news articles with topics. We'll cover three classification types: binary, multi-class, and multi-label.

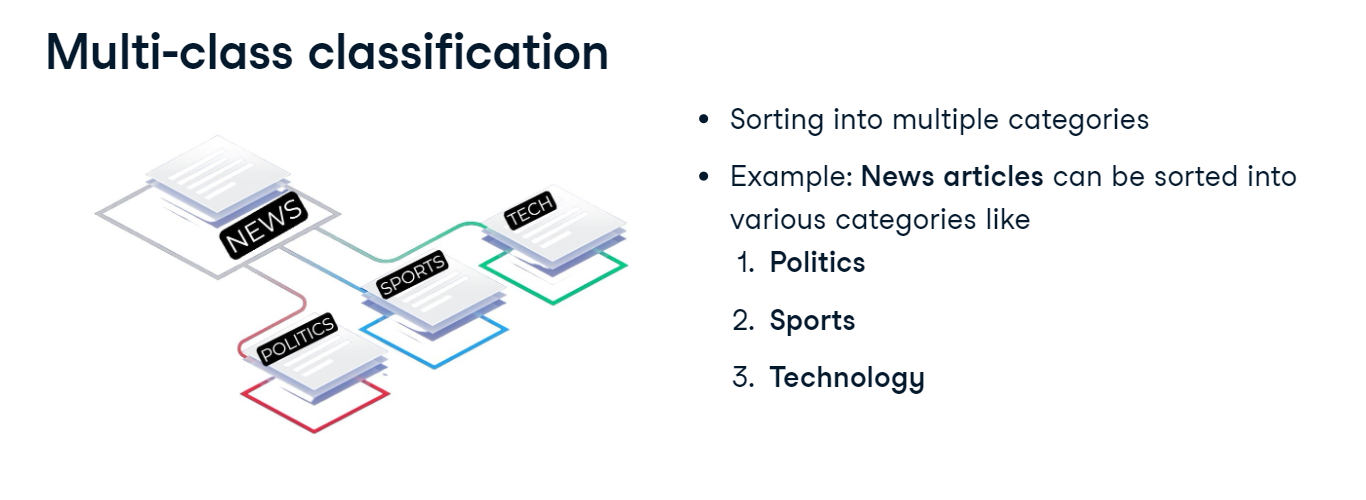
**Binary classification**

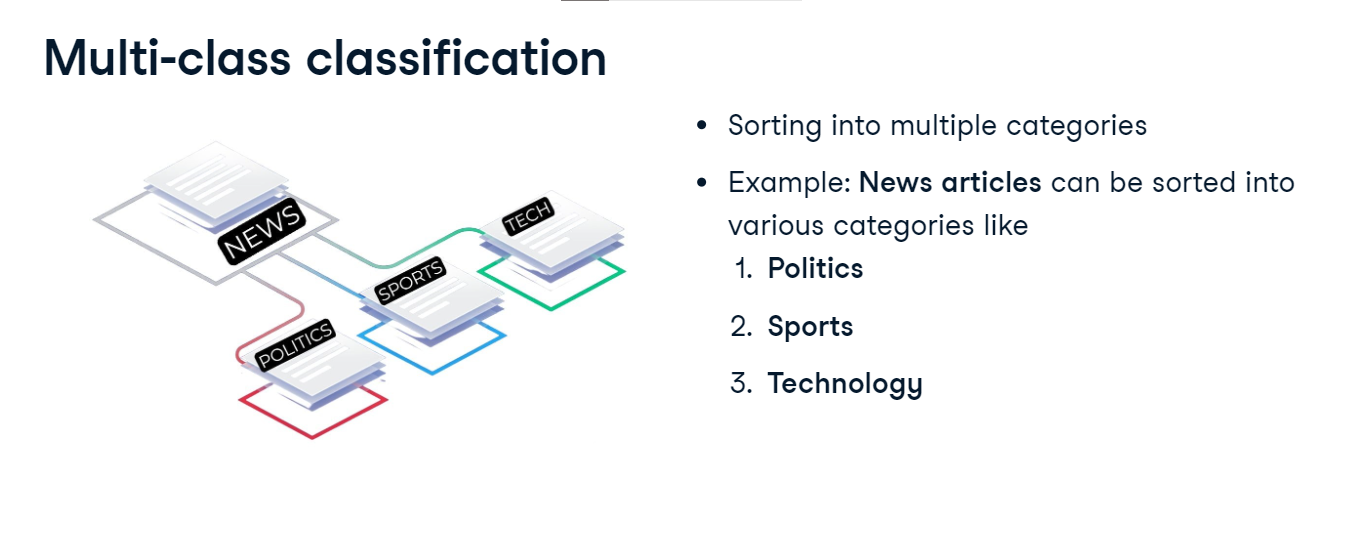
Binary classification sorts text into two categories, such as spam and not spam, as seen in email spam detection.

1. 1 <https://storage.googleapis.com/gweb-cloudblog-publish/images/image4_v2LFcq0.max-1200x1200.png>



**Multi-class classification**

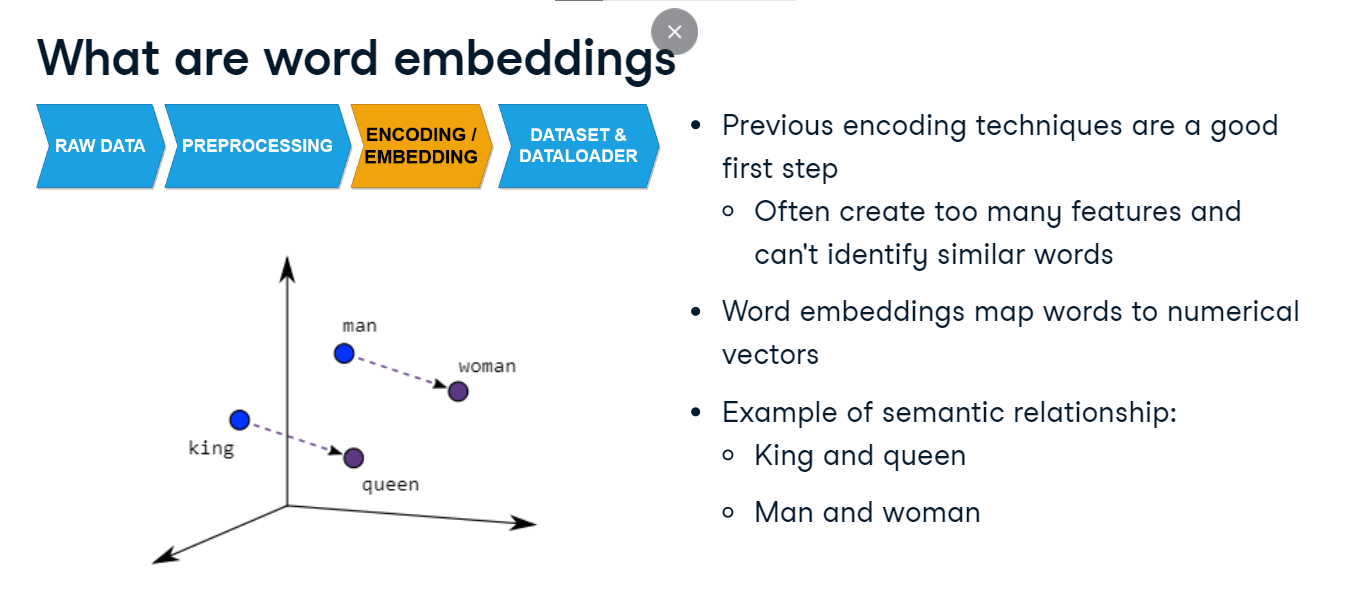
Multi-class classification categorizes text into more than two categories. For example, a news article could be classified into one of various categories, like politics, sports, or technology, depending on its content. 

**Multi-label classification**

In multi-label classification, text can belong to multiple categories simultaneously, unlike multi-class where it belongs to just one category. For example, a book can fit into multiple genres like action, adventure, and fantasy all at the same time.

**What are word embeddings**

Classifying text sometimes requires an understanding of the meaning of words. Previously, we covered encoding techniques including one-hot, bag-of-words, and TF-IDF in the processing pipeline. While these techniques are fundamental to preprocessing and are a good first step to extracting features, they often result in too many features and can't identify similar words. In contrast, word embeddings represent words as numerical vectors, preserving semantic meanings and connections between words like king and queen or man and woman. While the diagram shows three-dimensional vectors, real-world word embeddings often have much higher dimensionality.

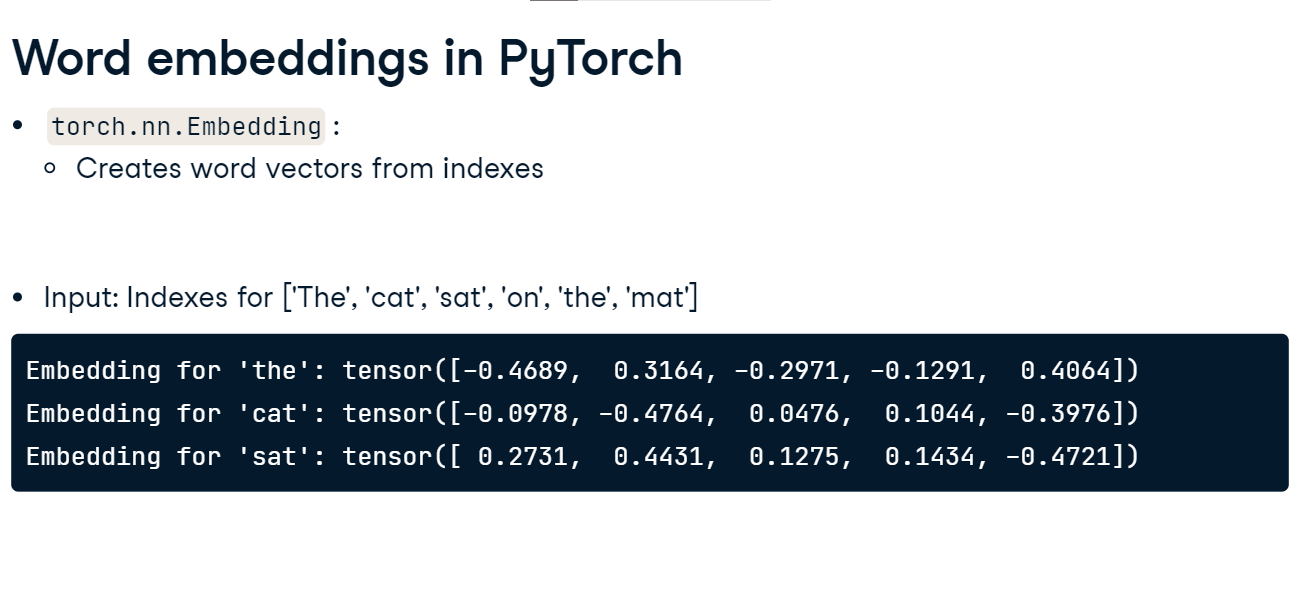


**Word to index mapping**

To enable embedding, we assign a unique index to a word with word-to-index mapping. For instance, we can translate "King" to one and "Queen" to two, giving us a numerical representation that is more compact and computationally efficient compared to the previous encoding techniques. Word-to-index mapping typically follows tokenization in the text processing pipeline, but it can follow any of the preproccessing techniques we've covered.

**Word embeddings in PyTorch**

PyTorch's torch-dot-nn-dot-Embedding is a flexible tool for creating word embeddings. It takes word indexes and transforms them into word vectors or embeddings. For example, given our sentence "The cat sat on the mat", it produces a unique vector for each word based on its index. Initially, these lists, or embeddings, contain random numbers because the model hasn't learned the meanings of the words yet. Through training, these embeddings start to change and learn, helping the model understand word meanings and their relationships.

****

**Using torch.nn.Embedding**

Let's implement word embeddings using PyTorch's nn-dot-Embedding. First, we construct our words list. We employ a dictionary, word-to-idx, to map words to indexes by enumerating the words through a dictionary comprehension. Utilizing the torch-dot-LongTensor function, we represent these mapped values as a tensor. We define an embedding layer with num\_embeddings argument set to the length of the words list. embedding\_dim specifies the size of each embedding vector; we've set it to ten. Remember, embedding\_dim is a hyperparameter that can be increased to tune our results further, but as we have only six words, we have chosen the smallest embedding. We create a tensor of indexes to pass through the embedding layer. The output is an embedding for each input value. In our case, it gives us a two-dimensional tensor with six rows and ten columns. Each row contains the embedding for the corresponding word.



**Using embeddings in the pipeline**

Here is what this step would look like in the full pipeline with the dataset and dataloader we previously created. Here, we perform our embedding on the data generated by the dataloader.

