Cheat Sheet: Generative AI Overview and Data Preparation

Package/Method	Description	Code example

NLTK	NLTK is a Python library used in natural language processing (NLP) for tasks such as tokenization and text processing.
	
	The code example shows how you can tokenize text using the NLTK word-based tokenizer.
		`import nltk`
		`nltk.download("punkt")`
		`from nltk.tokenize import word_tokenize`
		
		`text = "Unicorns are real. I saw a unicorn yesterday. I couldn't see it today."`
		`token = word_tokenize(text)`
		`print(token)`

spaCy	spaCy is an open-source library used in NLP. It provides tools for tasks such as tokenization and word embeddings.
	
	The code example shows how you can tokenize text using spaCy word-based tokenizer.
		`import spacy`
		`text = "Unicorns are real. I saw a unicorn yesterday. I couldn't see it today."`
		`nlp = spacy.load("en_core_web_sm")`
		`doc = nlp(text)`
		`token_list = [token.text for token in doc]`
		`print("Tokens:", token_list)`

BertTokenizer	BertTokenizer is a subword-based tokenizer that uses the WordPiece algorithm.
	
	The code example shows how you can tokenize text using BertTokenizer.
		`from transformers import BertTokenizer`
		
		`tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")`
		`tokenizer.tokenize("IBM taught me tokenization.")`

XLNetTokenizer	XLNetTokenizer tokenizes text using Unigram and SentencePiece algorithms.
	
	The code example shows how you can tokenize text using XLNetTokenizer.
		`from transformers import XLNetTokenizer`
		
		`tokenizer = XLNetTokenizer.from_pretrained("xlnet-base-cased")`
		`tokenizer.tokenize("IBM taught me tokenization.”)`

torchtext	The torchtext library is part of the PyTorch ecosystem and provides the tools and functionalities required for NLP.	`from torchtext.vocab import build_vocab_from_iterator`
		
		`# Defines a dataset`
		`dataset = [`

The code example shows how you can use torchtext to generate tokens and convert them to indices.

```
    (1,"Introduction to NLP"),
    (2,"Basics of PyTorch"),
    (1,"NLP Techniques for Text Classification"),
    (3,"Named Entity Recognition with PyTorch"),
    (3,"Sentiment Analysis using PyTorch"),
    (3,"Machine Translation with PyTorch"),
    (1,"NLP Named Entity,Sentiment Analysis, Machine 
Translation"),
    (1,"Machine Translation with NLP"),
    (1,"Named Entity vs Sentiment Analysis NLP")]

# Applies the tokenizer to the text to get the tokens as a 
list
from torchtext.data.utils import get_tokenizer

tokenizer = get_tokenizer("basic_english")
tokenizer(dataset[0][1])

# Takes a data iterator as input, processes text from the 
iterator, and yields the tokenized output individually
def yield_tokens(data_iter):
    for _,text in data_iter:
         yield tokenizer(text)

# Creates an iterator
my_iterator = yield_tokens(dataset)

# Fetches the next set of tokens from the data set
next(my_iterator)

# Converts tokens to indices and sets <unk> as the default 
word if a word is not found in the vocabulary
vocab = build_vocab_from_iterator(yield_tokens(dataset), 
specials=[“<unk>”])
vocab.set_default_index(vocab[“<unk>”])
```

		# Gives a dictionary that maps words to their corresponding numerical indices
		vocab.get_stoi()

vocab	The vocab object is part of the PyTorch torchtext library. It maps tokens to indices.
	
	The code example shows how you can apply the vocab object to tokens directly.
		# Takes an iterator as input and extracts the next tokenized sentence. Creates a list of token indices using the vocab dictionary for each token.
		
		def get_tokenized_sentence_and_indices(iterator):
		      tokenized_sentence = next(iterator)
		      token_indices = [vocab[token] for token in tokenized_sentence]
		      return tokenized_sentence, token_indices
		
		# Returns the tokenized sentences and the corresponding token indices. Repeats the process.
		
		tokenized_sentence, token_indices = \
		get_tokenized_sentence_and_indices(my_iterator)
		next(my_iterator)
		
		# Prints the tokenized sentence and its corresponding token indices.
		
		print("Tokenized Sentence:", tokenized_sentence)
		print("Token Indices:", token_indices)

Special tokens in PyTorch:
<eos> and <bos>
	Special tokens are tokens introduced to input sequences to convey specific information or serve a particular purpose during training.
	
	The code example shows the use of <bos> and <eos> during tokenization. The <bos> token denotes the beginning of the input sequence, and the <eos> token denotes the end.
		# Appends <bos> at the beginning and <eos> at the end of the tokenized sentences using a loop that iterates over the sentences in the input data
		
		tokenizer_en = get_tokenizer('spacy', language='en_core_web_sm')
		tokens = []
		max_length = 0
		
		for line in lines:

		tokenized_line = tokenizer_en(line)
		    tokenized_line = ['<bos>'] + tokenized_line + ['<eos>']
		    tokens.append(tokenized_line)
		    max_length = max(max_length, len(tokenized_line))

Special tokens in PyTorch: <pad>	The code example shows the use of <pad> token to ensure all sentences have the same length.	# Pads the tokenized lines for i in range(len(tokens)): tokens[i] = tokens[i] + ['<pad>'] * (max_length - len(tokens[i]))

Dataset class in PyTorch	The Dataset class enables accessing and retrieving individual samples from a data set.
	The code example shows how you can create a custom data set and access samples.
		# Imports the Dataset class and defines a list of sentences
		from torch.utils.data import Dataset
		
		sentences = ["If you want to know what a man's like, take a good look at how he treats his inferiors, not his equals.", "Fae's a fickle friend, Harry."]
		
		# Downloads and reads data
		class CustomDataset(Dataset):
		    def __init__(self, sentences):
		        self.sentences = sentences
		
		    # Returns the data length
		    def __len__(self):
		        return len(self.sentences)
		
		    # Returns one item on the index
		    def __getitem__(self, idx):
		        return self.sentences[idx]
		
		# Creates a dataset object
		dataset=CustomDataset(sentences)
		
		# Accesses samples like in a list
		E.g., dataset[0]

<table>
<tr><td>DataLoader class in PyTorch</td><td>A DataLoader class enables efficient loading and iteration over data sets for training deep learning models.

The code example shows how you can use the DataLoader class to generate batches of sentences for further processing, such as training a neural network model</td><td>

```
# Creates an iterator object
data_iter = iter(dataloader)

# Calls the next function to return new batches of samples
next(data_iter)
# Creates an instance of the custom data set
from torch.utils.data import DataLoader
custom_dataset = CustomDataset(sentences)

# Specifies a batch size
batch_size = 2

# Creates a data loader
dataloader = DataLoader(custom_dataset,
batch_size=batch_size, shuffle=True)

# Prints the sentences in each batch
for batch in dataloader:
    print(batch)
```

</td></tr>
<tr><td>Custom collate function in PyTorch</td><td>The custom collate function is a user-defined function that defines how individual samples are collated or batched together. You can utilize the collate function for tasks such as tokenization, converting tokenized indices, and transforming the result into a tensor.

The code example shows how you can use a custom collate function in a data loader.</td><td>

```
# Defines a custom collate function
def collate_fn(batch):
    tensor_batch = []

# Tokenizes each sample in the batch
    for sample in batch:
        tokens = tokenizer(sample)

# Maps tokens to numbers using the vocab
        tensor_batch.append(torch.tensor([vocab[token] for
token in tokens]))

# Pads the sequences within the batch to have equal lengths
    padded_batch =
pad_sequence(tensor_batch,batch_first=True)
    return padded_batch
```

</td></tr>
</table>

		`# Creates a data loader using the collate function and the custom dataset`
		`dataloader = DataLoader(custom_dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)`