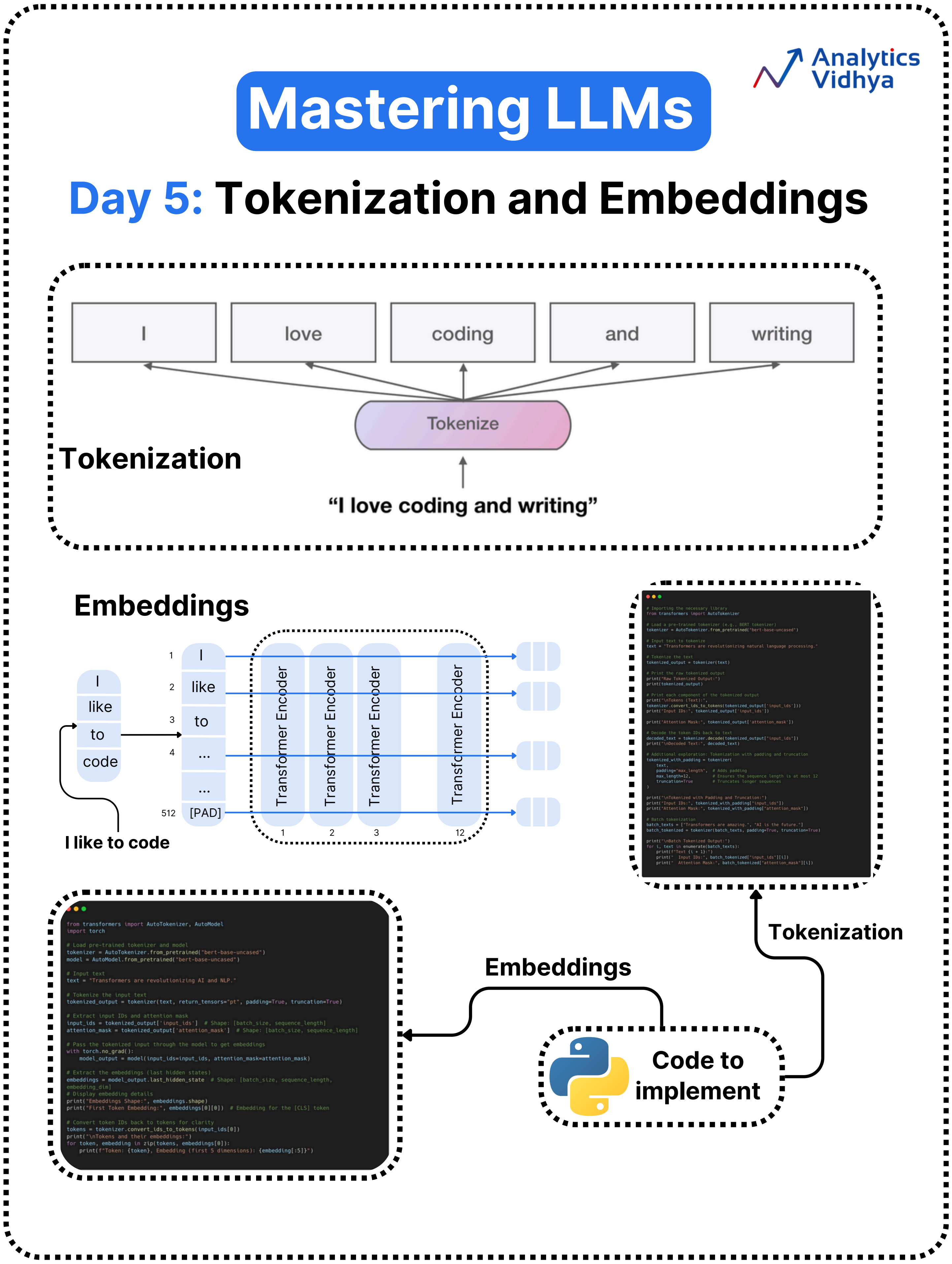
Mastering LLMs

Analytics Vidhya

Day 11: Word Emebeddings

```python
# Import necessary modules from the Hugging Face transformers and PyTorch libraries
from transformers import AutoTokenizer, AutoModel
import torch

# Step 1: Initialize the tokenizer
# Load the pre-trained tokenizer automatically for a given model name
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

# Step 2: Define a sample text to tokenize
sample_text = "Transformers are very powerful models for natural language processing."

# Step 3: Tokenize the input text
# Convert text to input tensors required for the model
inputs = tokenizer(sample_text, return_tensors="pt", padding=True, truncation=True)

# Step 4: Print the tokenized output
print("Tokenized Input IDs:", inputs['input_ids'])
print("Token Type IDs:", inputs['token_type_ids'])
print("Attention Mask:", inputs['attention_mask'])

# Step 5: Initialize the pre-trained model
model = AutoModel.from_pretrained('bert-base-uncased')

# Step 6: Pass the tokenized input to the model to get the embeddings
outputs = model(**inputs)

# Step 7: Extract the last hidden state (word embeddings for each token)
word_embeddings = outputs.last_hidden_state

# Step 8: Print the shape of the embeddings
# Shape explanation: (batch_size, sequence_length, hidden_size)
print("Word Embeddings Shape:", word_embeddings.shape)

# Step 9: Retrieve embeddings for individual words
# Example: Get the embeddings for the first token (CLS token)
cls_embedding = word_embeddings[0, 0, :]  # CLS token at index 0
print("CLS Token Embedding Shape:", cls_embedding.shape)

# Step 10: Convert embeddings to numpy for further processing if needed
cls_embedding_numpy = cls_embedding.detach().numpy()
print("CLS Token Embedding:", cls_embedding_numpy)
```

Up to this point, we've only explored the theoretical aspects; now, let's dive into implementing word embeddings in Python. Yesterday, we covered the implementation of tokenization in Python



Let’s learn how to implement word embeddings in Python

Loading the Pre-trained Tokenizer

AutoTokenizer.from_pretrained('bert-base-uncased') **loads the tokenizer** for the BERT model.

This tokenizer lowercases text and tokenizes it into subwords based on the model's vocabulary.

Tokenizing the Sample Text

tokenizer(sample_text, return_tensors="pt", padding=True, truncation=True) **converts text into token IDs.**

Outputs include:

- **input_ids**: Numerical token representations.
- **token_type_ids**: Token segmentation (used in tasks like QA).
- **attention_mask**: Marks padded tokens (1 for real tokens, 0 for padding).

Initializing the Pre-trained Model

AutoModel.from_pretrained('bert-base-uncased') loads the BERT model for **generating embeddings**.

Generating Word Embeddings

Passing tokenized inputs through the model produces the last hidden state, which contains **contextual embeddings** for each token in the sequence.

Understanding Word Embeddings Shape

outputs.last_hidden_state gives embeddings of shape (batch_size, sequence_length, hidden_size).

Example: If sequence_length=12 and hidden_size=768, the shape will be (1, 12, 768).

Extracting Specific Embeddings

[0, 0, :] extracts the [CLS] token, which acts as a representation for the entire sentence.

Converting to Numpy for Further Use

.detach().numpy() is used to convert the PyTorch tensor to a NumPy array for downstream processing.

Output Example (Sample Execution Output)

```
Tokenized Input IDs: tensor([[ 101, 19081,  2024,  2200,  2843,  4274,  2005,  3012,  2653,  3973,
  1012,   102]])
Token Type IDs: tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
Attention Mask: tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
Word Embeddings Shape: torch.Size([1, 12, 768])
CLS Token Embedding Shape: torch.Size([768])
CLS Token Embedding: [-0.2034  0.3182 ... -0.2156]
```

Key Applications of Word Embeddings

- **Sentence Classification**: Using [CLS] embedding for sentiment analysis or topic detection.
- **Named Entity Recognition** (NER): Utilizing token-level embeddings for identifying entities in text.
- **Semantic Similarity**: Comparing embeddings of different texts to measure similarity.
- **Downstream NLP Tasks**: Fine-tuning for tasks like translation, summarization, etc.

Analytics Vidhya

Stay Tuned for **Day 12** of

Mastering LLMs