Sample Code

Finding answers to questions in a text document

Locate relevant passages in a document by asking the Bidirectional Encoder Representations from Transformers (BERT) model a question.

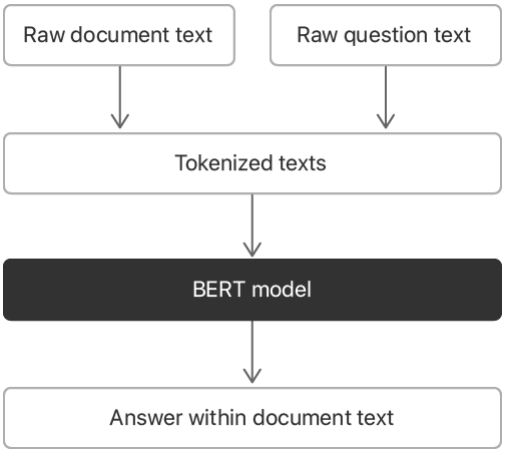
Download

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | Xcode 15.2+

Overview

This sample app leverages the BERT model to find the answer to a user's question in a body of text. The model accepts text from a document and a question, in natural English, about the document. The model responds with the location of a passage within the document text that answers the question. For example, given the text, "The quick brown fox jumps over the lethargic dog.", with the question "Who jumped over the dog?", the BERT model's predicted answer is, "the quick brown fox".

The BERT model does not generate new sentences to answer a given question. It finds the passage in a document that's most likely to answer the question.



The sample leverages the BERT model by:

1. Importing the BERT model's vocabulary into a dictionary
2. Breaking up the document and question texts into tokens
3. Converting the tokens to ID numbers using the vocabulary dictionary
4. Packing the converted token IDs into the model's input format
5. Calling the BERT model's prediction(from:) method
6. Locating the answer by analyzing the BERT model's output
7. Extracting that answer from the original document text

Configure the sample code project

Before you run the sample code project in Xcode, use a device with either:

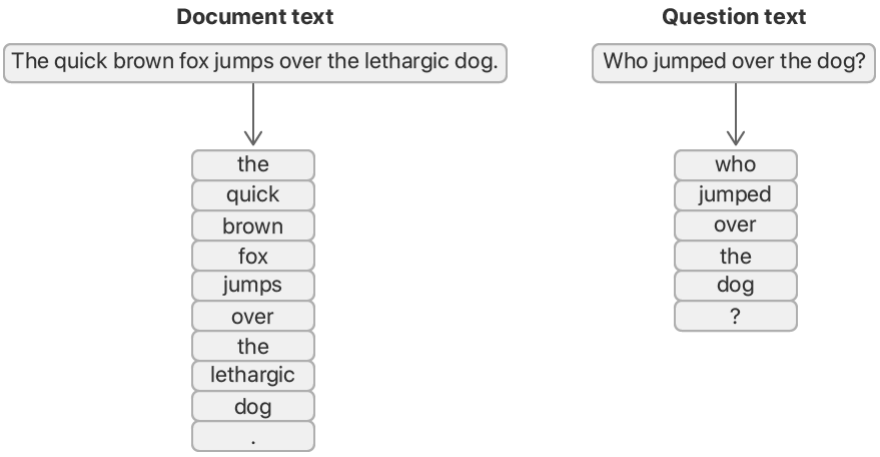
- iOS 13 or later
- macOS 10.15 or later

Build the vocabulary

The first step to using the BERT model is to import its vocabulary. The sample creates a vocabulary dictionary by splitting the vocabulary file into lines, each of which has one token. The function assigns each token's (zero-based) line number as its value. For example, the first token, `"[PAD]"`, has an ID of `0`, and the 5,001st token, `"knight"`, has an ID of `5000`.

Split the text into word tokens

The BERT model requires you to convert each word into one or more token IDs. Before you can use the vocabulary dictionary to find those IDs, you must divide the document's text and the question's text into word tokens.



The sample does this by using an NLTagger, which breaks up a string into word tokens, each of which is a substring of the original.

```
// Store the tokenized substrings into an array.
var wordTokens = [Substring]()

// Use Natural Language's NLTagger to tokenize the input by word.
let tagger = NLTagger(tagSchemes: [.tokenType])
tagger.string = rawString

// Find all tokens in the string and append to the array.
tagger.enumerateTags(in: rawString.startIndex..<rawString.endIndex,
                     unit: .word,
                     scheme: .tokenType,
                     options: [.omitWhitespace]) { (_, range) -> Bool in
    wordTokens.append(rawString[range])
    return true
}

return wordTokens
```

The sample app leverages the tagger to split each string into tokens by using its enumerateTags(in:unit:scheme:options:using:) method with the .tokenType tagging scheme and the .word token unit.

Convert word or wordpiece tokens into their IDs

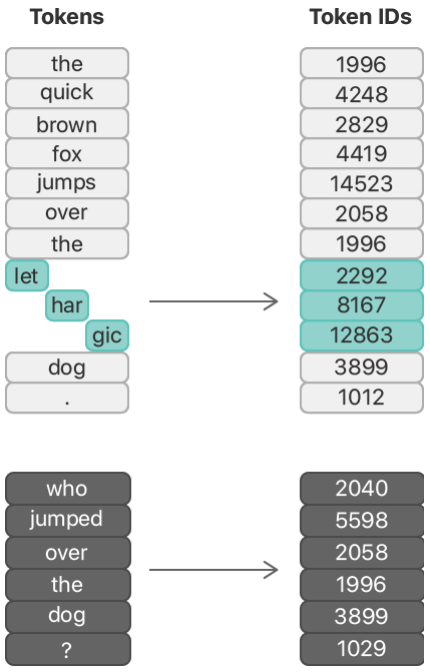
For speed and efficiency, the BERT model operates on token IDs, which are numbers that represent tokens, rather than operating on the text tokens themselves.

```
let subTokenID = BERTVocabulary.tokenID(of: searchTerm)
```

If a word token doesn't exist in the vocabulary, the method looks for subtokens, or *wordpieces*. A wordpiece is a component of a larger word token. For example, the word *lethargic* isn't in the vocabulary but its wordpieces, *let*, *har*, and *gic* are. Dividing the vocabulary's large words into wordpieces reduces the vocabulary size and makes the BERT model more flexible. The model can understand words that aren't explicitly in the vocabulary by combining their wordpieces.

Secondary wordpieces, such as *har* and *gic*, each appear in the vocabulary with two leading pound signs, as `##har` and `##gic`.

Continuing the example, the method converts document text into the word and wordpiece token IDs shown in the following figure.



Prepare the model input

The BERT model has two inputs:

- `wordIDs` — Accepts the document and question texts
- `wordTypes` — Tells the BERT model which elements of `wordIDs` are from the document

The sample creates the `wordIDs` array by arranging the token IDs in the following order:

1. A *classification start* token ID, which has a value of `101` and appears as `"[CLS]"` in the vocabulary file
2. The token IDs from the question string
3. A *separator* token ID, which has a value of `102` and appears as `"[SEP]"` in the vocabulary file
4. The token IDs from the text string
5. Another separator token ID

6. One or more *padding* token IDs for the remaining, unused elements, which have a value of `0` and appear as `"[PAD]"` in the vocabulary file

```
// Start the wordID array with the `classification start` token.
var wordIDs = [BERTVocabulary.classifyStartTokenID]

// Add the question tokens and a separator.
wordIDs += question.tokenIDs
wordIDs += [BERTVocabulary.separatorTokenID]

// Add the document tokens and a separator.
wordIDs += document.tokenIDs
wordIDs += [BERTVocabulary.separatorTokenID]

// Fill the remaining token slots with padding tokens.
let tokenIDPadding = BERTInput.maxTokens - wordIDs.count
wordIDs += Array(repeating: BERTVocabulary.paddingTokenID, count: tokenIDPadding)
```

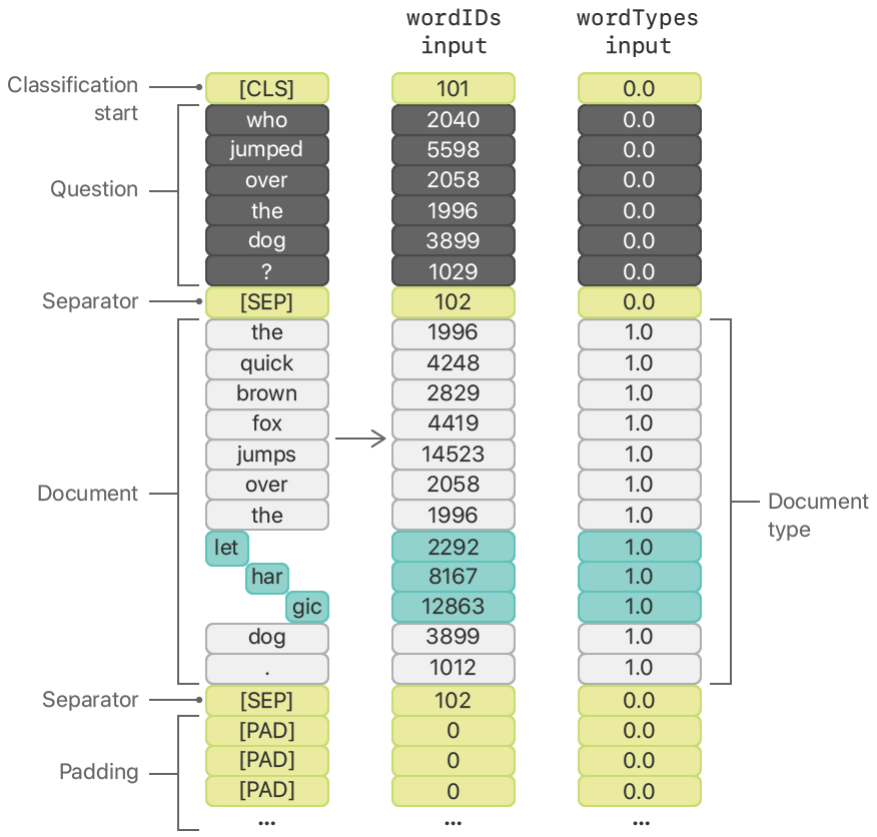
Next, the sample prepares the `wordTypes` input by creating an array of the same length, where all the elements that correspond to the document text are `1` and all others are `0`.

```
// Set all of the token types before the document to 0.
var wordTypes = Array(repeating: 0, count: documentOffset)

// Set all of the document token types to 1.
wordTypes += Array(repeating: 1, count: document.tokens.count)

// Set the remaining token types to 0.
let tokenTypePadding = BERTInput.maxTokens - wordTypes.count
wordTypes += Array(repeating: 0, count: tokenTypePadding)
```

Continuing the example, the sample arranges the two input arrays with the values shown in the figure below.



Next, the sample creates an `MLMultiArray` for each input and copies the contents from the arrays, which it uses to create a `BERTQAFP16Input` feature provider.

> **Note**
>
> The BERT model in this sample requires a one-dimensional `MLMultiArray` input with 384 elements. Models from other sources may have different inputs or shapes.

```
// Create the MLMultiArray instances.
let tokenIDMultiArray = try? MLMultiArray(wordIDs)
let wordTypesMultiArray = try? MLMultiArray(wordTypes)

// Unwrap the MLMultiArray optionals.
guard let tokenIDInput = tokenIDMultiArray else {
    fatalError("Couldn't create wordID MLMultiArray input")
}

guard let tokenTypeInput = wordTypesMultiArray else {
    fatalError("Couldn't create wordType MLMultiArray input")
}

// Create the BERT input MLFeatureProvider.
let modelInput = BERTQAFP16Input(wordIDs: tokenIDInput,
                                 wordTypes: tokenTypeInput)
```

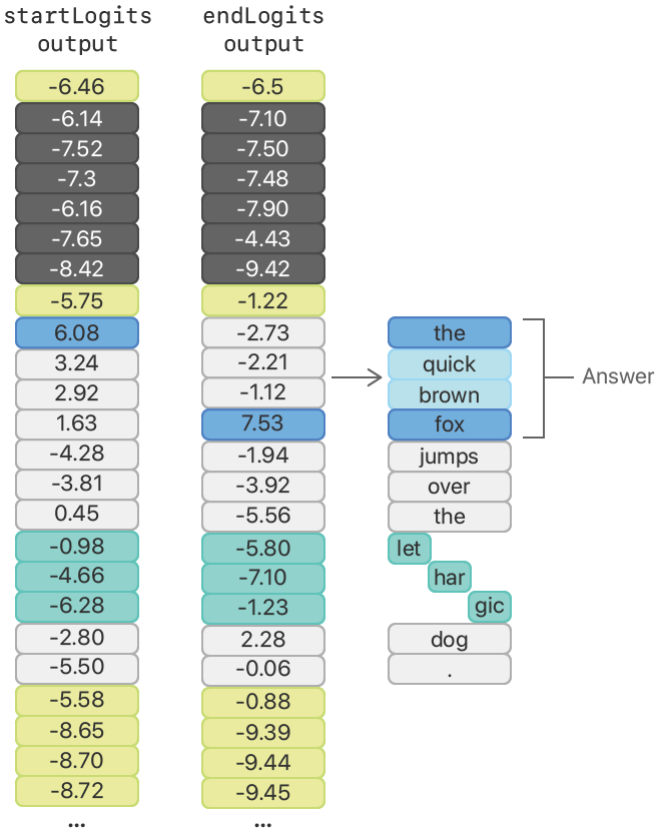
Make a prediction

You use the BERT model to predict where to find an answer to the question in the document text, by giving the model your input feature provider with the input `MLMultiArray` instances.

```
guard let prediction = try? bertModel.prediction(input: modelInput) else {
    return "The BERT model is unable to make a prediction."
}
```

Find the answer

You locate the answer to the question by analyzing the output from the BERT model. The model produces two outputs, `startLogits` and `endLogits`. Each *logit* is a raw confidence score of where the BERT model predicts the beginning and the end of an answer is.



In this example, the best start and end logits are `6.08` and `7.53` for the tokens `"the"` and `"fox"`, respectively. The sample finds the indices of the highest-value starting and ending logits by:

1. Converting each output logit `MLMultiArray` into a `Double` array.

2. Isolating the logits relevant to the document.

3. Finding the indices, in each array, to the 20 logits with the highest values.

4. Searching through the 20 x 20 or fewer combinations of logits for the best combination.

```
// Convert the logits MLMultiArrays to [Double].
let startLogits = prediction.startLogits.doubleArray()
let endLogits = prediction.endLogits.doubleArray()

// Isolate the logits for the document.
let startLogitsOfDoc = [Double](startLogits[range])
let endLogitsOfDoc = [Double](endLogits[range])

// Only keep the top 20 (out of the possible ~380) indices for faster searching.
let topStartIndices = startLogitsOfDoc.indicesOfLargest(20)
let topEndIndices = endLogitsOfDoc.indicesOfLargest(20)

// Search for the highest valued logit pairing.
let bestPair = findBestLogitPair(startLogits: startLogitsOfDoc,
                                 bestStartIndices: topStartIndices,
                                 endLogits: endLogitsOfDoc,
                                 bestEndIndices: topEndIndices)
```

In this example, the indices of the best start and end logits are 8 and 11, respectively. The answer substring, located between indices 8 and 11 of the original text, is `"the quick brown fox"`.

Scale for larger documents

The BERT model included in this sample can process up to 384 tokens, including the three overhead tokens—one "classification start" token and two separator tokens—leaving 381 tokens for your text and question, combined. For larger texts that exceed this limitation, consider using one of these techniques:

- Use a search mechanism to narrow down the relevant document text.
- Break up the document text into sections, such as by paragraph, and make a prediction for each section.