Sahil Gangele

Prof. ChengXiang Zhai

CS 410 - Text Information Systems

November 6, 2021

Technology Review: Apple's Natural Language Framework

The Natural Language framework was introduced by Apple during their WWDC (world wide developers conference) 2018 event, showcasing how iOS, iPadOS, tvOS, macOS, and watchOS applications could use on-device API's to perform powerful Natural language processing (NLP) tasks written in Swift. This was a significant milestone for the Apple ecosystem as their mobile hardware as two large constraints: memory and power. NLP tasks typically can be costly to memory by storing large collections of documents or costly to the battery by performing expensive computation to compare large documents to each other. By creating a suite of API's for developers to easily perform NLP tasks, Apple paved the way for each of its devices to become more intelligent. This paper will discuss the various functions that can be performed in Apple's Natural Language (NL) framework, and will perform a detailed look into the concept of word embeddings and how it can be used to perform intelligent comparisons among text. Among the first concepts we'll look at is language detection.

Apple has retail stores in more than 25 countries and currently the App Store is available in 175 regions all across the world. Therefore, NLP tasks on Apple devices must have breadth in their languages they support, and depth in their understanding of each language. In doing so, NLP currently supports 57 languages. To identify languages in one's application, Apple recommends working with `NLLanguageRecognizer` to observe the dominant language in a string of text.

Tokenization is the basis of all NLP tasks, as its the way we normalize lexical units into the same form. This is important, and difficult to do in a standard way for all languages as for every language one cannot assume spaces in between words are the delimiters to tokenize each word. Languages such as Chinese and Japanese won't work in these cases. `NLTokenizer` is the API Apple provides and is basis for all tokenization tasks. `NLTokenizer` can be modified using various lexical units as well (word, sentences, paragraphs etc.).

Part of Speech (POS) tagging allows us to analyze the syntactical categories of words. For example, in the sentence "A dog is chasing a boy on the playground", "dog" would be a noun, "chasing" is a verb, "playground" is a noun etc. Why is this useful? Say Siri, the intelligent voice assistant for Apple devices, has to understand what a command is. The command could be asking a question, or it could be asking Siri to do something, such as "Add a reminder". Knowing the POS tags allows us to figure out whether a verb is in a sentence and if it is, then how we would go about acting on that verb. Utilize `NLTagger` and set the `tagSchemes` to `.lexicalClass` in order to obtain POS tagging. To perform lemmatization, simply change the tagSchemes to `.lemma`.

Semantic analysis in NLP is a difficult task to perform, as it requires deeper NLP and even more human effort. However, it brings us closer to our own knowledge representation. We

see this used in many areas across iOS, from a location being underlined in a message bubble in the Messages app, to being able to long press a persons name in an email to bring up their contact info. Utilize the `NLTagger` class with the `tagSchemes` of `.nameType` in order to retrieve entities from text.

Determining whether a certain lexical unit: word, sentence, or paragraph is saying something positive or negative is a useful metric when analyzing text. It essentially helps us solve the problem of classification for text into two categories: good or bad. Apple provides an API to perform sentiment analysis and will return a sentiment score between -1 (bad) to 1 (good). To utilize this API, use the `NLTagger` class with the `tagSchemes` of `.sentimentScore` to retrieve sentiment score from text.

Apple released the ability to work with word embeddings on iOS 12, and was embedded in all OS's supporting 7 different languages. Word embeddings is a new phenomenon that is an improvement over the Bag of Words (BOW) representation of text. Word embeddings considers the context of words around it in order to make a better decision of comparing whether two lexical units are similar to each other. This is an improvement over the BOW representation which took each word to be independent from one another. In word embeddings, we use the vector space model to represent as each word as a vector. These vectors are simply a continuous sequence of numbers that can have N dimensions. These values for the word embeddings are dependent on a text corpus that Apple uses which contains billions of words. The assumption word embeddings makes is if two words occur in similar contexts, then they are also similar words. Essentially, the similarity of two words is defined by the context in which a word occurs. If we plot these vectors, objects that are similar semantically are also clustered together. With these vectors, we can perform 4 basic functionalities with word embeddings: obtaining a vector for word, computing the distance between two words (eg. "Dog" and "Cat" should be closer than "Dog" and "Car"), getting the nearest neighbors for word (eg. For "Dog" we would get "Breeds", "Puppies", "Walks" etc.) and lastly obtaining the nearest neighbors for vector (eg. For a sentence, sum up all the vectors for each word, then ask NL Framework for all the words closest to this vector). Apple even lets developers bring their own custom word embeddings for applications that work within a specific domain and use unique vocabulary (medical field for eg.). Apple supports Word2Vec, GloVe, fasttext and any custom neural networks to be embedded in the NL Framework. The unique part about Apple's use of word embeddings is how they compress large 1 or 2 gb embeddings into a compact couple of dozen mb format. Specifically, a 2 gb word embedding model from GloVe when converted into being used with the NL Framework is able to compress it to 31 mb. Also, it optimizes the time at which it takes to perform the nearest neighbors computation to nearly a couple of milliseconds. When it comes to measuring similarity of two words, Apple uses cosine similarity and will return a value between 0.0 and 2.0, 0.0 being identical and 2.0 having no similarity. To utilize this, Apple recommends working with the `NLEmbedding` object. This provides functions to compute the distance between two vectors and the ability to find the nearest k neighbors to a vector. Thus far, we've only talked about the idea of a specific type of word embeddings called Static Word Embeddings. It's called static due to the vector for each word being precomputed based on a text corpus Apple used. This makes for a quick lookup, however static word embeddings have a one main drawback. There is a lack of word coverage due to a finite language set being used. If a word in the text is not seen in the finite language of the static word embedding model, then

an accurate similarity score cannot be computed. To solve this problem, Apple introduced Sentence Word Embeddings in iOS 13. With this, when we pass in a sentence into the NL framework, it dynamically analyzes the entire sentence and encodes it into a finite dimensional vector. Since we choose our lexical unit of choice to be a sentence, our assumption now changes to two vectors being similar if two sentences are conceptually similar. To accomplish this task was fairly complicated on Apple's end, however to use the API is simple. Utilize `NLEmbedding.sentenceEmbedding` to work directly with Apple's sentence embedding technology.

Apple's Natural Language Framework is built, optimized, and used by and for Apple devices. This is apparent, from adopting multiple language support in NL framework for their global reach of users to the space optimization in memory constrained devices with word embeddings. We've discussed multiple features of the NL Framework including Language Detection, Tokenization, Parts of Speech (POS) Tagging, Named Entity Recognition, Sentiment Analysis and finally a closer look into the concept of word embeddings and how Apple allows us to take advantage of this new technology. Overall, Apple's NL Framework provides a robust set of API's and capabilities to increase intelligence in apps across all of Apple's devices. This is apparent as all API's in the NL Framework discussed are performed on device, performant, optimized for Swift and not sent to the cloud which ensures one of the pillars of Apple's viewpoints: privacy.

Coding Examples

This is a follow up to all the NL Framework API's talked about above. All examples are custom made to show the various capabilities of the NL Framework.

Language Detection

```
import NaturalLanguage

// This is Hindi, translated to english meaning "How are you?"
let text = "आप कैसे हैं"
// Create the language recognizer
let langaugeRecognizer = NLLanguageRecognizer()
// Process the string
langaugeRecognizer.processString(text)
// Obtain the dominant language
let dominantLanguage = langaugeRecognizer.dominantLanguage!.rawValue

print(dominantLanguage) // Prints "Hi" which is Hindi in ISO 639-1 code
```

Tokenization

```
import NaturalLanguage

let text = "My name is Sahil. I am 24 years old."
let tokenizer = NLTokenizer(unit: .sentence)
tokenizer.string = text
tokenizer.enumerateTokens(in: text.startIndex..<text.endIndex) { index, attributes in
    print(text[index])
    return true
}
// This prints out:
// My name is Sahil.
// I am 24 years old.

// You can change the unit to be a `.word` then the following output would be:
// My, name, is, Sahil, I, am, 24, years, old
```

POS Tagging

```
import NaturalLanguage

let text = "A dog is chasing a boy on the playground."
let tagger = NLTagger(tagSchemes: [.lexicalClass])
tagger.string = text

tagger.enumerateTags(in: text.startIndex..<text.endIndex, unit: .word, scheme: .lexi-
calClass, options: [.omitPunctuation, .omitWhitespace]) { tag, index in
    if let tag = tag {
        print("\(text[index]) : \(tag.rawValue)")
    }
    return true
}

// Prints the following:, A : Determiner, dog : Noun, is : Verb, chasing : Verb, a :
Determiner, boy : Noun, on : Preposition, the : Determiner, playground : Noun
```

Lemmatization

```
import NaturalLanguage

let text = "A dog is chasing a boy on the playground."
let tagger = NLTagger(tagSchemes: [.lemma])
tagger.string = text

tagger.enumerateTags(in: text.startIndex..<text.endIndex, unit: .word, scheme: .lemma,
options: [.omitPunctuation, .omitWhitespace]) { tag, index in
    if let tag = tag {
        print("\(text[index]) : \(tag.rawValue)")
    }
    return true
}

// This prints:, dog : dog, is : be, chasing : chase, a : a, boy : boy, on : on, the :
the, playground : playground
```

Handing Ambiguities of the word “Design”

```
import NaturalLanguage

let text = "Can you design me a shirt? Actually, what is the design of that shirt?"
let tagger = NLTagger(tagSchemes: [.lexicalClass])
tagger.string = text

tagger.enumerateTags(in: text.startIndex..<text.endIndex, unit: .word, scheme: .lexi-
calClass, options: [.omitPunctuation, .omitWhitespace]) { tag, index in
    if let tag = tag {
        print("\(text[index]) : \(tag.rawValue)")
    }
    return true
}
// Prints: Can : Verb, you : Pronoun, design : Verb, me : Pronoun, a : Determiner,
shirt : Noun, Actually : Adverb, what : Pronoun, is : Verb, the : Determiner, design :
Noun, of : Preposition, that : Determiner, shirt : Noun
```

Named Entity Recognition

```
import NaturalLanguage

let text = "Hi, my name is Sahil Gangele. And I live in Ashburn, Virginia"
let tagger = NLTagger(tagSchemes: [.nameType])
let tags: [NLTag] = [.personalName, .placeName]
tagger.string = text

tagger.enumerateTags(in: text.startIndex..<text.endIndex, unit: .word, scheme: .name-
Type, options: [.omitPunctuation, .omitWhitespace, .joinNames]) { tag, index in
    if let tag = tag, tags.contains(tag) {
        print("\(text[index]) : \(tag.rawValue)")
    }
    return true
}
/* Prints:
Sahil Gangele : PersonalName
Ashburn : PlaceName
Virginia : PlaceName
*/
```

Named Entity Recognition

```
import NaturalLanguage

let tagger = NLTagger(tagSchemes: [.sentimentScore])
let text = "Zelda Breath of the Wild is the best video game I have ever played!"
tagger.string = text
let (sentiment, _) = tagger.tag(at: text.startIndex, unit: .paragraph, scheme: .senti-
mentScore)
print(sentiment!.rawValue)
// Prints 0.8, meaning it's confident it's a positive statement
```

Word Embeddings - Distance Function

```
import NaturalLanguage

let embedding = NLEmbedding.wordEmbedding(for: .english)!

let catDogDistance = embedding.distance(between: "cat", and: "dog")
let catCarDistance = embedding.distance(between: "cat", and: "car")
print(catDogDistance) // Prints 0.7168956398963928, which means it's fairly close
print(catCarDistance) // Prints 1.2544564008712769, which means the two words are more
orthogonal
```

Word Embeddings - Nearest Neighbors

```
import NaturalLanguage

let embedding = NLEmbedding.wordEmbedding(for: .english)!
let cat = "cat"
for (word, confidence) in embedding.neighbors(for: cat, maximumCount: 5) {
    print("\(word): \(confidence)")
}
/* Prints
feline: 0.6218327879905701
kitten: 0.6540808081626892
dog: 0.7168956398963928
tabby: 0.7377545237541199
pet: 0.7584871053695679
*/
```

Sentence Embeddings - Distance Function

```
import NaturalLanguage

let embedding = NLEmbedding.sentenceEmbedding(for: .english)!

let sentenceOne = "This game is not fun"
let sentenceTwo = "This game is very fun"

let distance = embedding.distance(between: sentenceOne, and: sentenceTwo)
print(distance) // Prints 0.9343676567077637 because "not fun" and "very fun" are dif-
ferent orthongonal concepts

let sentenceThree = "This game is kinda fun"
let newDistance = embedding.distance(between: sentenceTwo, and: sentenceThree)
print(newDistance) // Prints 0.4919203519821167, which makes sense as "kinda" and
"very" fun are similar.
```