Artificial Intelligence Using Swift

by Mark Watson

Artificial Intelligence Using Swift

CoreML, NLP, Deep Learning, Semantic Web and Linked Data, Knowledge Graphs, Knowledge Representation

Mark Watson

This book is available at https://leanpub.com/SwiftAI

This version was published on 2025-06-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Cover Material, Copyright, and License

This eBook will be updated occasionally so please periodically check the leanpub.com web page for this book[1] for updates.

The first edition was released spring of 2022. The second edition was released December 2024. The third edition was released June 2025.

If you would like to support my work please consider purchasing my books on Leanpub[2] and star my git repositories that you find useful on GitHub[3]. You can also interact with me on social media on Mastodon[4] and Twitter[5].

[1] https://leanpub.com/SwiftAI
[2] https://leanpub.com/u/markwatson
[3] https://github.com/mark-watson?tab=repositories&q=&type=public
[4] https://mastodon.social/@mark_watson
[5] https://twitter.com/mark_l_watson

Preface

Why use Swift for hacking AI? Common Lisp has been my go-to language for artificial intelligence development and research since 1982. The transition to using Swift was a transition motivated by practical aspects of Swift and the Swift ecosystem.

Notes on the new June 2025 Book Edition

With the release of new Apple Intelligence tooling available for macOS26, iPadOS26, and iOS26 I have added a new Part 5 at the end of this book covering several practical AI use cases with new examples.

Requests from the Author

This book will always be available to read free online at https://leanpub.com/SwiftAI/read[6].

That said, I appreciate it when readers purchase my books because the income enables me to spend more time writing.

Hire the Author as a Consultant

I am available for short consulting projects. Please see https://markwatson.com[7].

You can also interact with me on social media on Mastodon[8] and Twitter[9].

Notes on the Second Edition

The second edition of this book deletes some of the old material and adds two new themes:

- A new Part II of the book that covers Large Language Models (LLMS). We will use both commercial LLM APIs and running local LLMs using Ollama and Apple's MLX framework.
- Several examples from the first edition are augmented using LLMs.
- As much as possible, I support some of the book examples as Swift Playgrounds, usable on iPads and Macs.

[6] https://leanpub.com/SwiftAI/read
[7] https://markwatson.com
[8] https://mastodon.social/@mark_watson
[9] https://twitter.com/mark_l_watson

Book Structure

This book starts out slowly in Part I with simple examples which I wrote showing how to access the Swift library packages on GitHub, tips on writing Swift command line apps.

Part II will show you how to effectively integrate LLMs into your own applications.

Part III starts with a simple example using web scraping and commercial web search APIs. We then work through examples integrating web search with LLMs and then show how we can modify web scraping applications to specifically process topics and have better control of outputing structured data.

We then proceed to using Apple's CoreML for Natural Language Processing (NLP), training and using your own CoreML models, using OpenAI's GPT-4 APIs, and finally several semantic web/linked data examples. The book ends with the example macOS application Knowledge Graph Navigator. It is not my intention to cover in detail the use of SwiftUI for building iOS/iPadOS/macOS applications but I thought my readers might enjoy seeing several of the techniques covered in the book integrated into an example app.

I have used Common Lisp for AI research projects and for AI product development and delivery since 1982. There is something special about using a language for almost forty years. I now find Swift a compelling choice for several reasons:

- Flexible language with many features I rely on like supporting closures and an interactive functional programming style.
- Built-in support for deep learning neural network models for natural language processing, predictive models, etc.
- First class support for iOS and macOS development.
- Good support for server side applications hosted on Linux.

Swift is a programmer-efficient language: code is concise and easy to read, and high quality libraries from Apple and third parties mean that often there is less code to write. I will share with you my Swift development work flow that combines interactive development of code in playgrounds, development of higher level libraries in text only or command line applications, and my general strategy for writing iOS and macOS applications after low level and intermediate code is written and debugged.

Requests from the Author

This book will always be available to read free online at https://leanpub.com/SwiftAI/read[10].

That said, I appreciate it when readers purchase my books because the income enables me to spend more time writing.

[10] https://leanpub.com/SwiftAI/read

Hire the Author as a Consultant

I am available for short consulting projects. Please see https://markwatson.com[11].

Parts of this Book are Specific for macOS and iOS, with Some Support for Linux

Swift is a general purpose language that is well supported in macOS, iOS, and Linux, with some support in Windows. Here, we cover the use of Swift on macOS and iOS. Some of the examples in this book rely on libraries that are specifically available on macOS and iOS like CoreML and the NLP libraries. Several book examples also work on Linux, such as the examples using SQLite, the Microsoft Azure search APIs, web scraping, and semantic web/linked data.

Code for this Book

Because of the way the Swift Package Manager works, I organized all book examples that build libraries as separate GitHub repos so the libraries can be easily used in other book examples as well as your own software projects. The separate library GitHub repositories are:

- https://github.com/mark-watson/SparqlQuery_swift[12] - SPARQL Swift library for my Swift AI book.
- https://github.com/mark-watson/QuestionAnswering_BERT_swift[13] - modification of Apple's question answering demo to use DBPedia.
- https://github.com/mark-watson/swift-coreml-wisconsin_data_create_model[14] - create CoreML models from training data files of Wisconsin Caner data.
- https://github.com/mark-watson/swift-coreml-wisconsin_data_predict_with_model[15] - use the pretrained Wisconsin Cancer data model.
- https://github.com/mark-watson/ShellProcess_swift[16] - library for spawning shell processes and capturing output to stdout.
- https://github.com/mark-watson/WebScraping_swift[17] - library for scrapping web sites.
- https://github.com/mark-watson/OpenAI_swift[18] - library for using OpenAI's GPT3 APIs.
- https://github.com/mark-watson/Nlp_swift[19] - library that uses pretrained CoreML NLP models.

[11] https://markwatson.com
[12] https://github.com/mark-watson/SparqlQuery_swift
[13] https://github.com/mark-watson/QuestionAnswering_BERT_swift
[14] https://github.com/mark-watson/swift-coreml-wisconsin_data_create_model
[15] https://github.com/mark-watson/swift-coreml-wisconsin_data_predict_with_model
[16] https://github.com/mark-watson/ShellProcess_swift
[17] https://github.com/mark-watson/WebScraping_swift
[18] https://github.com/mark-watson/OpenAI_swift
[19] https://github.com/mark-watson/Nlp_swift

- https://github.com/mark-watson/KGN[20] - SwiftUI based application supporting macOS, iPadOS, and iOS. The macOS version is in Apple's app store.

I suggest cloning all of these GitHub repositories right now so you can have the example source code at hand while reading this book.

Except for the last SwiftUI example application, all sample programs are written as command line utilities. I considered using Swift playgrounds for some of the examples but decided that packaging as a combination of libraries and command line utilities would tend to make the example code more useful for your own projects.

http://www.knowledgegraphnavigator.com/

Author's Background

I have written 20+ books, mostly about artificial intelligence. I have over 50 US patents.

I write about technologies that I have used throughout my career: knowledge representation using semantic web and linked data, machine learning and deep learning, and natural language processing. I am grateful for the companies where I have worked (SAIC, Google, Capital One, Olive AI, Babylist, etc.) that have supported this work since 1982.

As an author, I hope that the material in this book entertains you and will be useful in your work.

Cover Art

The cover picture was taken by WikiMedia Commons user Keta[21] and is available for use under the Creative Commons License CC BY-SA 2.5.

CoreML Libraries Used in this Book

- CoreML general overview: https://developer.apple.com/documentation/coreml
- MLClassifier https://developer.apple.com/documentation/createml/mlclassifier
- MLTextClassifier https://developer.apple.com/documentation/createml/mltextclassifier
- NLModel https://developer.apple.com/documentation/naturallanguage/nlmodel
- Natural Language Framework https://developer.apple.com/documentation/naturallanguage
- MLCustomLayer https://developer.apple.com/documentation/coreml/mlcustomlayer

[20] https://github.com/mark-watson/KGN
[21] https://commons.wikimedia.org/wiki/User:Keta

Swift 3rd Party Libraries

We use the following 3rd party libraries:

- https://github.com/SwiftyJSON/SwiftyJSON[22]

Acknowledgements

I thank my wife Carol for editing this manuscript, finding typos, and suggesting improvements.

[22] https://github.com/SwiftyJSON/SwiftyJSON

Part 1: Introduction and Short Examples

We begin with a sufficient introduction for Swift to understand the programming examples. After introducing the language we will look at a few short examples that provide code and techniques we use later in the book:

- Creating Swift projects
- Writing command line utilities
- Web scraping

Setting Up Swift for Command Line Development

Except for the last chapter in this book that uses Xcode for developing a complete macOS/iOS/iPadOS example application, I assume that you will work through the book examples using the command line and your favorite editor. If you want to use Xcode for the command line examples, you can open the Swift package file on the command line and open Xcode using, for example:

```
cd SparqlQuery_swift
open Package.swift
```

You notice that most of the examples are command line apps or libraries with command line test programs and the **README.md** files in the example directories provide instructions for building and running on the command line.

You can also run Xcode and from the File Menu open an example **Package.swift** file. You can then use the Product / Test menu to run the test code for the example. You might need to use the View / Debug Area / Active Console menu to show the output area.

I assume that you are familiar with the Swift programming language and Xcode.

Swift is a general purpose language that is well supported in macOS and iOS, with good support for Linux, and with some support in Windows. For the purposes of this book, we are only considering the use of Swift on macOS and iOS. Most of the examples in this book rely on libraries that are specifically available on macOS and iOS like CoreML and the NLP libraries.

There are great free resources for the Swift language on the web, in other commercial books, and Apple's free Swift books. Here I provide just enough material on the Swift language for you to understand and work with the book examples. After working through this book's material you will be able to add machine learning, natural language processing, and knowledge representation to your applications. There will be parts of the Swift language that we don't need for the material here, and we won't cover.

Installing Swift Packages

We will use the Swift Package Manager[23]. You should pause reading now and install the Swift Package Manager if you have not already done so.

[23] https://swift.org/package-manager/

I occasionally use https://vapor.codes web framework[24] library (although not in this book). We use this 3rd party library as an example for building a library locally from source code. Start by cloning the git repository https://github.com/vapor/vapor[25]. Then:

```
git clone https://github.com/vapor/vapor.git
cd vapor
swift build
```

I don't usually install libraries locally from source code unless I am curious about the implementation and want to read through the source code. Later we will see how to reference Swift libraries hosted on GitHub in a project's **Package.swift** file.

Creating Swift Packages

We will cover using the Swift Package Manager to create new packages using the command line here. Later we will create projects using Apple's XCode IDE when we develop the example application Knowledge Graph Navigator.

You will want to use the Swift Package Manager documentation[26] for reference.

We will be generating executable projects and library (with a sample main program) projects. The commands for generating the stub for an executable application project are:

```
mkdir BingSearch
cd BingSearch
swift package init --type executable
```

and build the stub of a library with a demo main program:

```
mkdir SparqlQuery
cd SparqlQuery
swift package init --type library
```

Accessing Libraries that You Write in Other Projects

You can reference Swift libraries using the **Swift.package** file for each of your projects. We will look at parts of two **Swift.package** files here. The first is for my SPARQL query client library that we will develop in a later chapter. This library **SparqlQuery_swift** is used in both book examples Knowledge Graph Navigator (**KGN**) macOS/iOS/iPadOS example application as well as a text version **KnowledgeGraphNavigator_swift**.

[24] https://vapor.codes
[25] https://github.com/vapor/vapor
[26] https://github.com/apple/swift-package-manager/blob/main/Documentation/Usage.md

```
import PackageDescription

let package = Package(
    name: "SparqlQuery_swift",
    products: [
        .library(
            name: "SparqlQuery_swift",
            targets: ["SparqlQuery_swift"]),
    ],
    dependencies: [
      .package(url: "https://github.com/SwiftyJSON/SwiftyJSON.git",
          .branch("master")),
   ],
    targets: [
        .target(
            name: "SparqlQuery_swift",
            dependencies: ["SwiftyJSON"]),
        .testTarget(
            name: "SparqlQuery_swiftTests",
            dependencies: ["SparqlQuery_swift", "SwiftyJSON"]),
    ]
)
```

This Swift package file is used to declare a Swift package named "SparqlQuery_swift". The package contains one library target named "SparqlQuery_swift" and one test target named "SparqlQuery_-swiftTests". The library target depends on the "SwiftyJSON" package, which is specified as a dependency in the "dependencies" section of the package.

The "products" section defines the products that this package provides. In this case, the package provides a library product named "SparqlQuery_swift". The library is built from the source code in the "SparqlQuery_swift" target.

The "dependencies" section lists the packages that this package depends on. In this case, it depends on the "SwiftyJSON" package, which is specified as a Git repository URL.

The "targets" section lists the targets that are part of the package. In this case, there are two targets: "SparqlQuery_swift" and "SparqlQuery_swiftTests". The "SparqlQuery_swift" target depends on "SwiftyJSON". The "SparqlQuery_swiftTests" target depends on both "SparqlQuery_swift" and "SwiftyJSON".

The **Swift.package** file for text version **KnowledgeGraphNavigator_swift** is shown here:

```
import PackageDescription

let package = Package(
    name: "KnowledgeGraphNavigator_swift",
    platforms: [
        .macOS(.v10_15),
    ],
    dependencies: [
        .package(url: "https://github.com/SwiftyJSON/SwiftyJSON.git",
            .branch("master")),
        .package(url: "https://github.com/scinfu/SwiftSoup.git", from: "1.7.4"),
        .package(url: "git@github.com:mark-watson/SparqlQuery_swift.git",
            .branch("main")),
        .package(url: "git@github.com:mark-watson/Nlp_swift.git", .branch("main")),
    ],
    targets: [
        // Targets are the basic building blocks of a package.
        // A target can define a module or a test suite.
        // Targets can depend on other targets in this package,
        // and on products in packages this package depends on.
        .target(
            name: "KnowledgeGraphNavigator_swift",
            dependencies: ["SparqlQuery_swift", "Nlp_swift",
              "SwiftyJSON", "SwiftSoup"]),
    ]
)
```

This Swift package file is used to declare a Swift package named “KnowledgeGraphNavigator_swift”. The package contains one target named “KnowledgeGraphNavigator_swift”. The target depends on the “SparqlQuery_swift”, “Nlp_swift”, “SwiftyJSON”, and “SwiftSoup” packages, which are specified as dependencies in the “dependencies” section of the package.

The “platforms” section specifies the minimum platform version that the package supports. In this case, the package supports macOS version 10.15 and later.

The “dependencies” section lists the packages that this package depends on. In this case, it depends on four packages:

- SwiftyJSON: a Swift library for working with JSON data.
- SwiftSoup: a Swift library for parsing HTML and XML documents.
- SparqlQuery_swift: a Swift library for querying RDF data using the SPARQL query language.
- Nlp_swift: a Swift library for natural language processing.

The “targets” section lists the targets that are part of the package. In this case, there is one target named **KnowledgeGraphNavigator_swift**. The target depends on “**SparqlQuery_swift**, **Nlp_swift**, **SwiftyJSON**, and **SwiftSoup**.

Hopefully you have cloned the git repositories for each book example and understand how I have configured the examples for your use.

For the rest of this book, you can read chapters in any order. In some cases, earlier chapters will contain implementations of libraries used in later chapters.

Background Information for Writing Swift Command Line Utilities

This short chapter contains example code and utilities for writing command line programs, using external shell processes, and using the FileIO library.

Using Shell Processes

The library for using shell processes is one of my GitHub projects so you can include it in other projects using:

```
 dependencies: [
   .package(url: "git@github.com:mark-watson/ShellProcess_swift.git",
            .branch("main")),
 ],
```

You can clone this repository if you want to have the source code at hand:

```
git clone https://github.com/mark-watson/ShellProcess_swift.git
```

The following listing shows the library implementation. In line 5 we use the constructor **Process** from the Apple **Foundation** library to get a new process object that we set fields **executableURL** and **argList**. In lines 8 and 9 we create a new Unix style pipe to capture the output from the shell process we are starting and attach it to the process. After we run the task, we capture the output and return it as the value of function **run_in_shell**.

```
import Foundation

@available(OSX 10.13, *)
public func run_in_shell(commandPath: String, argList: [String] = []) -> String {
    let task = Process()
    task.executableURL = URL(fileURLWithPath: commandPath)
    task.arguments = argList
    let pipe = Pipe()
    task.standardOutput = pipe
    do {
```

```
11          try! task.run()
12          let data = pipe.fileHandleForReading.readDataToEndOfFile()
13          let output: String? = String(data: data, encoding: String.Encoding.utf8)
14          if let output = output {
15            if !output.isEmpty {
16              return output.trimmingCharacters(in: .whitespacesAndNewlines)
17            }
18          }
19      }
20      return ""
21  }
```

The function named **run_in_shell** takes two parameters: **commandPath** (a string representing the path to the executable command to be run) and **argList** (an array of strings representing the arguments to be passed to the command). The function returns a string that represents the output of the command.

Function **run_in_shell** first creates an instance of the **Process** class, which is used to run the command. It sets the **executableURL** property of the task instance to the **commandPath** value and sets the arguments property to the argList value. This function then creates a Pipe instance, which is used to capture the output of the command. It sets the standardOutput property of the task instance to the Pipe instance.

The function then runs the command using the **run()** method of the task instance. If the command runs successfully, the function reads the output of the command from the **Pipe** instance using the **readDataToEndOfFile()** method of the **fileHandleForReading** property. It then converts the output data to a string using the **String(data:encoding:)** initializer.

If the output string is not empty, this function trims leading and trailing whitespace and returns the resulting string. Otherwise, the function returns an empty string.

Overall, this function provides a simple way to run a shell command and capture its output in a Swift program.

As in most examples in this book we use the Swift testing framework to run the example code at the command line using *swift test*. Running *swift test* does an implicit *swift build*.

```
import XCTest
@testable import ShellProcess_swift

final class ShellProcessTests: XCTestCase {
    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the
        // correct results.
        print("** s1:")
        let s1 = run_in_shell(commandPath: "/bin/ps", argList: ["a"])
        print(s1)
        let s2 = run_in_shell(commandPath: "/bin/ls", argList: ["."])
        print("** s2:")
        print(s2)
        let s3 = run_in_shell(commandPath: "/bin/sleep", argList: ["2"])
        print("** s3:")
        print(s3)

    }

    static var allTests = [
        ("testExample", testExample),
    ]
}
```

This Swift unit test function is part of a test suite for the **ShellProcess_swift** package. The function is named **testExample** and is decorated with the **@testable** import statement to indicate that it tests an internal implementation detail of the **ShellProcess_swift** package.

The function uses the **run_in_shell** function to run three shell commands: **ps a**, **ls .**, and **sleep 2**. It prints the output of each command to the console.

This test function is an example of a functional test case. It doesn't actually verify that the functions being tested produce the correct results. Instead, it's a simple way to visually inspect the output of the commands and ensure that they are working as expected.

The **allTests** variable is an array of tuples that map the test function names to the corresponding function references. This variable is used by the XCTest framework to discover and run the test functions.

The test output (with some text removed for brevity) is:

```
$ swift test
Test Suite 'All tests' started at 2021-08-06 16:36:21.447
** s1:
PID   TT  STAT      TIME COMMAND
 3898 s000  Ss     0:00.01 login -pf markw8
 3899 s000  S+     0:00.18 -zsh
 3999 s001  Ss     0:00.02 login -pfl markw8 /bin/bash -c exec -la zsh /bin/zsh
 4000 s001  S+     0:00.38 -zsh
 5760 s002  Ss     0:00.02 login -pfl markw8 /bin/bash -c exec -la zsh /bin/zsh
 5761 s002  S      0:00.14 -zsh
 8654 s002  S+     0:00.06 /Applications/Xcode.app/Contents/Developer/Toolchains/Xco\
deDefault.xctoolchain/usr/bin/swift-test
 8665 s002  S      0:00.03 /Applications/Xcode.app/Contents/Developer/usr/bin/xctest\
 /Users/markw_1/GIT_swift_book/ShellProcess_swift/.build/arm64-apple-macosx/debug/Sh\
ellProcess_swiftPackageTests.xctest
 8666 s002  R      0:00.00 /bin/ps a
** s2:
Package.swift
README.md
Sources
Tests
** s3:

Test Suite 'All tests' passed at 2021-08-06 16:36:23.468.
         Executed 1 test, with 0 failures (0 unexpected) in 2.019 (2.021) seconds
```

FileIO Examples

This file I/O example uses the **ShellProcess_swift** library we saw in the last section so if you were to create your own Swift project with the following code listing, you would have to add this dependency in the **Project.swift** file.

When writing command line Swift programs you will often need to do simple file IO so let's look at some examples here:

```
import Foundation
import ShellProcess_swift // my library

@available(OSX 10.13, *)
func test_files_demo() -> Void {
    // In order to append to an existing file, you need to get a file handle
    // and seek to the end of a file. The following will not work:
    let s = "the dog chased the cat\n"
    try! s.write(toFile: "out.txt", atomically: true,
                 encoding: String.Encoding.ascii)
    let s2 = "a second string\n"
    try! s2.write(toFile: "out.txt", atomically: true,
                  encoding: String.Encoding.ascii)
    let aString = try! String(contentsOfFile: "out.txt")
    print(aString)

    // For simple use cases, simply appending strings, then writing
    // the result atomically works fine:
    var s3 = "the dog chased the cat\n"
    s3 += "a second string\n"
    try! s3.write(toFile: "out2.txt", atomically: true,
                  encoding: String.Encoding.ascii)
    let aString2 = try! String(contentsOfFile: "out2.txt")
    print(aString2)

    // list files in current directory:
    let ls = run_in_shell(commandPath: "/bin/ls", argList: ["."])
    print(ls)

    // remove two temnporary files:
    let shellOutput = run_in_shell(commandPath: "/bin/rm",
                                   argList: ["out.txt", "out2.txt"])
    print(shellOutput)
}

if #available(OSX 10.13, *) {
    test_files_demo()
}
```

The OS version checks in this Swift code use the **#available** conditional compilation block.

The **#available block** is used to conditionally compile code based on the availability of APIs or features in the operating system version. In this case, the code inside the **#available(OSX 10.13, *)** block will only be executed if the running operating system is macOS 10.13 or later.

If the running operating system version is earlier than 10.13, the code inside the **#available block** will be skipped and the program will exit without running the **test_files_demo()** function.

These operating system version checks are done to ensure that the program is only executed on operating systems that support the APIs and features used by the code. This helps to prevent runtime errors and crashes on older operating system versions that may not support the required features.

This function demonstrates how to write to and read from files using the **write(toFile:atomically:encoding:)** and **String(contentsOfFile:)** methods, how to list files in the current directory using the ls shell command, and how to remove files using the rm shell command.

I created a temporary Swift project with the previous code listing and a **Project.swift** file. I built and ran this example using the **swift** command line tool.

Unlike the example in the last section where we built a reusable library with a test program, here we have a standalone program contained in a single file so we will use *swift run* to build and run this example:

```
$ swift run
Fetching git@github.com:mark-watson/ShellProcess_swift.git from cache
Cloning git@github.com:mark-watson/ShellProcess_swift.git
Resolving git@github.com:mark-watson/ShellProcess_swift.git at main
[5/5] Build complete!
a second string

the dog chased the cat
a second string

Package.resolved
Package.swift
README.md
Sources
out.txt
out2.txt
```

Swift REPL

There is an example of using the Swift REPL at the end of the next chapter on web scraping. For reference, you can start a REPL with:

```
$ swift run --repl
Type :help for assistance.
1> import WebScraping_swift
2> webPageText(uri: "https://markwatson.com")
$R0: String = "Mark Watson: AI Practitioner and Polyglot Programmer"...
3> public func foo(s: String) -> String { return s }
4> foo(s: "cat")
$R1: String = "cat"
5>
```

You can import packages and interactively enter Swift expressions, including defining functions.

In the next chapter we will look at a longer example that scrapes web sites.

In the next chapter we will look at one more simple example, building a web scraping library, before getting to the machine learning and NLP part of the book.

Web Scraping

It is important to respect the property rights of web site owners and abide by their terms and conditions for use. This Wikipedia article on Fair Use[27] provides a good overview of using copyright material.

The web scraping code we develop here uses the Swift library **SwiftSoup** that is loosely based on the BeautifulSoup libraries available in other programming languages.

For my work and research, I have been most interested in using web scraping to collect text data for natural language processing but other common applications include writing AI news collection and summarization assistants, trying to predict stock prices based on comments in social media which is what we did at Webmind Corporation in 2000 and 2001, etc.

I wrote a simple web scraping library that is available at https://github.com/mark-watson/Web-Scraping_swift[28] that you can use in your projects by putting the following dependency in your **Project.swift** file:

```
    dependencies: [
         .package(url: "git@github.com:mark-watson/WebScraping_swift.git",
             .branch("main")),
    ],
```

Here is the main implementation file for the library:

```
import Foundation
import SwiftSoup

public func webPageText(uri: String) -> String {
    guard let myURL = URL(string: uri) else {
        print("Error: \(uri) doesn't seem to be a valid URL")
        fatalError("invalid URI")
    }
    let html = try! String(contentsOf: myURL, encoding: .ascii)
    let doc: Document = try! SwiftSoup.parse(html)
    let plain_text = try! doc.text()
    return plain_text
}

```

[27] https://en.wikipedia.org/wiki/Fair_use
[28] https://github.com/mark-watson/WebScraping_swift

```
15 func webPageHeadersHelper(uri: String, headerName: String) -> [String] {
16     var ret: [String] = []
17     guard let myURL = URL(string: uri) else {
18         print("Error: \(uri) doesn't seem to be a valid URL")
19         fatalError("invalid URI")
20     }
21     do {
22         let html = try String(contentsOf: myURL, encoding: .ascii)
23         let doc: Document = try SwiftSoup.parse(html)
24         let h1_headers = try doc.select(headerName)
25         for el in h1_headers {
26             let h1 = try el.text()
27             ret.append(h1)
28         }
29     } catch {
30         print("Error")
31     }
32     return ret
33 }
34 
35 public func webPageH1Headers(uri: String) -> [String] {
36     return webPageHeadersHelper(uri: uri, headerName: "h1")
37 }
38 
39 public func webPageH2Headers(uri: String) -> [String] {
40     return webPageHeadersHelper(uri: uri, headerName: "h2")
41 }
42 
43 public func webPageAnchors(uri: String) -> [[String]] {
44     var ret: [[String]] = []
45     guard let myURL = URL(string: uri) else {
46         print("Error: \(uri) doesn't seem to be a valid URL")
47         fatalError("invalid URI")
48     }
49     do {
50         let html = try String(contentsOf: myURL, encoding: .ascii)
51         let doc: Document = try SwiftSoup.parse(html)
52         let anchors = try doc.select("a")
53         for a in anchors {
54             let text = try a.text()
55             let a_uri = try a.attr("href")
56             if a_uri.hasPrefix("#") {
57                 ret.append([text, uri + a_uri])
```

```
58              } else {
59                  ret.append([text, a_uri])
60              }
61          }
62      } catch {
63          print("Error")
64      }
65      return ret
66  }
```

This Swift code defines several functions that can be used to scrape information from a web page located at a given URI.

The **webPageText** function takes a URI as input and returns the plain text content of the web page located at that URI. It first checks if the URI is valid and then reads the content of the web page using the contentsOf method of the String class. It then uses the parse method of the SwiftSoup library to parse the HTML content of the page and extract the plain text.

The **webPageH1Headers** and **webPageH2Headers** functions use the **webPageHeadersHelper** function to extract the **H1** and **H2** header texts respectively from the web page located at a given URI. The **webPageHeadersHelper** function uses the same technique as the **webPageText** function to read and parse the HTML content of the page. It then selects the headers using the specified **headerName** parameter and extracts the text of the headers.

The **webPageAnchors** function extracts all the anchor tags **<a>** from the web page located at a given URI, along with their corresponding text and URI. It also uses the **webPageHeadersHelper** function to read and parse the HTML content of the page, selects the anchor tags using the "**a**" selector, and extracts their **text** and **href** attributes.

Overall, these functions provide a simple way to scrape information from a web page and extract specific information such as plain text, header texts, and anchor tags.

I wrote these utility functions to get the plain text from a web site, HTML header text, and anchors. You can clone this library and extend it for other types of HTML elements you may need to process.

The test program shows how to call the APIs in the library:

```
1  import XCTest
2  import Foundation
3  import SwiftSoup
4
5  @testable import WebScraping_swift
6
7  final class WebScrapingTests: XCTestCase {
8      func testGetWebPage() {
9          let text = webPageText(uri: "https://markwatson.com")
```

```
        print("\n\n\tTEXT FROM MARK's WEB SITE:\n\n", text)
    }

    func testToShowSwiftSoupExamples() {
        let myURLString = "https://markwatson.com"
        let h1_headers = webPageH1Headers(uri: myURLString)
        print("\n\n++ h1_headers:", h1_headers)
        let h2_headers = webPageH2Headers(uri: myURLString)
        print("\n\n++ h2_headers:", h2_headers)
        let anchors = webPageAnchors(uri: myURLString)
        print("\n\n++ anchors:", anchors)
}

    static var allTests = [("testGetWebPage", testGetWebPage),
                           ("testToShowSwiftSoupExamples",
                            testToShowSwiftSoupExamples)]
}
```

This Swift test program tests the functionality of the **WebScraping_swift** library. It defines two test functions: **testGetWebPage** and **testToShowSwiftSoupExamples**.

The **testGetWebPage** function uses the **webPageText** function to retrieve the plain text content of my website located at "https://markwatson.com". It then prints the retrieved text to the console.

The **testToShowSwiftSoupExamples** function demonstrates the use of **webPageH1Headers**, **webPageH2Headers**, and **webPageAnchors** functions on the same website. It extracts and prints the H1 and H2 header texts and anchor tags of the same website.

The **allTests** variable is an array of tuples that map the test function names to the corresponding function references. This variable is used by the XCTest framework to discover and run the test functions.

Overall, this Swift test program demonstrates how to use the functions defined in the WebScraping_swift library to extract specific information from a web page.

Here we run the unit tests (with much of the output not shown for brevity):

```
$ swift test

        TEXT FROM MARK's WEB SITE:

 Mark Watson: AI Practitioner and Polyglot Programmer | Mark Watson    Read my Blog \
   Fun stuff    My Books    My Open Source Projects    Hire Me    Free Mentoring    \
Privacy Policy Mark Watson: AI Practitioner and Polyglot Programmer I am the author \
of 20+ books on Artificial Intelligence, Common Lisp, Deep Learning, Haskell, Clojur\
e, Java, Ruby, Hy language, and the Semantic Web. I have 55 US Patents. My customer \
list includes: Google, Capital One, Olive AI, CompassLabs, Disney, SAIC, Americast, \
PacBell, CastTV, Lutris Technology, Arctan Group, Sitescout.com, Embed.ly, and Webmi\
nd Corporation.

++ h1_headers: ["Mark Watson: AI Practitioner and Polyglot Programmer", "The books t\
hat I have written", "Fun stuff", "Open Source", "Hire Me", "Free Mentoring", "Priva\
cy Policy"]

++ h2_headers: ["I am the author of 20+ books on Artificial Intelligence, Common Lis\
p, Deep Learning, Haskell, Clojure, Java, Ruby, Hy language, and the Semantic Web. I\
 have 55 US Patents.", "Other published books:"]

++ anchors: [["Read my Blog", "https://mark-watson.blogspot.com"], ["Fun stuff", "ht\
tps://markwatson.com#fun"], ["My Books", "https://markwatson.com#books"], ["My Open \
Source Projects", "https://markwatson.com#opensource"], ["Hire Me", "https://markwat\
son.com#consulting"], ["Free Mentoring", "https://markwatson.com#mentoring"], ["Priv\
acy Policy", "https://markwatson.com/privacy.html"], ["leanpub", "https://leanpub.co\
m/u/markwatson"], ["GitHub", "https://github.com/mark-watson"], ["LinkedIn", "https:\
//www.linkedin.com/in/marklwatson/"], ["Twitter", "https://twitter.com/mark_l_watson\
"], ["leanpub", "https://leanpub.com/lovinglisp"], ["leanpub", "https://leanpub.com/\
haskell-cookbook/"], ["leanpub", "https://leanpub.com/javaai"],
]
Test Suite 'All tests' passed at 2021-08-06 17:37:11.062.
         Executed 2 tests, with 0 failures (0 unexpected) in 0.471 (0.472) seconds
```

Running in the Swift REPL

```
$ swift run --repl
[1/1] Build complete!
Launching Swift REPL with arguments: -I/Users/markw_1/GIT_swift_book/WebScraping_swi\
ft/.build/arm64-apple-macosx/debug -L/Users/markw_1/GIT_swift_book/WebScraping_swift\
/.build/arm64-apple-macosx/debug -lWebScraping_swift__REPL
Welcome to Apple Swift version 5.5 (swiftlang-1300.0.29.102 clang-1300.0.28.1).
Type :help for assistance.
  1> import WebScraping_swift
  2> webPageText(uri: "https://markwatson.com")
$R0: String = "Mark Watson: AI Practitioner and Polyglot Programmer | Mark Watson   \
 Read my Blog    Fun stuff    My Books    My Open Source Projects    Privacy Policy \
Mark Watson: AI Practitioner and Polyglot Programmer I am the author of 20+ books on\
 Artificial Intelligence, Common Lisp, Deep Learning, Haskell, Clojure, Java, Ruby, \
Hy language, and the Semantic Web. I have 55 US Patents. My customer list includes: \
Google, Capital One, Babylist, Olive AI, CompassLabs, Disney, SAIC, Americast, PacBe\
ll, CastTV, Lutris Technology, Arctan Group, Sitescout.com, Embed.ly, and Webmind Co\
rporation"...
  3>
```

This chapter finishes a quick introduction to using Swift and Swift packages for command line utilities. The remainder of this book comprises machine learning, natural language processing, and semantic web/linked data examples.

Part 2: Large Language Models

In this part we cover:

- Commercial OpenAI LLM APIs
- Commercial Anthropic LLM APIs
- Accessing open weight models using the commercial Groq service
- Accesiing xAIs Grok model via an API
- Accessing local LLMs using Ollama
- Using Local LLMs with Apple's MLX Framework

Using the OpenAI LLM APIs

I have been working as an artificial intelligence practitioner since 1982 and the capability of Large Language Models (LLMs) is unlike anything I have seen before. I managed a deep learning team at Capital One in 2017-2019 and we used precursors of TransFormer models like OpenAI's ChatGPT, and Anthropic's Claude.

You will need to apply to OpenAI for an access key at:

```
https://platform.openai.com/signup
```

The GitHub repository for this example is:

```
https://github.com/mark-watson/OpenAI_swift
```

I recommend reading the online documentation for the online documentation for the APIs[29] to see all the capabilities of the beta OpenAI APIs. Let's start by jumping into the example code that is a GitHub repository https://github.com/mark-watson/OpenAI_swift[30] that you can use in your projects.

The library that I wrote for this chapter supports four functions: for completing text, summarizing text, answering general questions, and getting embeddings for text. The get-4o-mini that we will use here is very inexpensive and capable.

You need to request an API key (I had to wait a few weeks to receive my key) and set the value of the environment variable **OPENAI_KEY** to your key. You can add a statement like:

```
export OPENAI_KEY=sa-hdedds7&dhdhsdffd...
```

to your .**profile** or other shell resource file that contains your key value (the above key value is made-up and invalid).

The file **Sources/OpenAI_swift/OpenAI_swift.swift** contains the source code (code description follows the listing):

[29] https://openai.com/docs/
[30] https://github.com/mark-watson/OpenAI_swift

```
import Foundation

struct OpenAI {
    private static let key = ProcessInfo.processInfo.environment["OPENAI_KEY"]!
    private static let baseURL = "https://api.openai.com/v1"

    private struct ChatRequest: Encodable {
        let model: String
        let messages: [[String: String]]
        let max_tokens: Int
        let temperature: Double
    }

    private struct EmbeddingRequest: Encodable {
        let model: String
        let input: String
    }

    private static func makeRequest<T: Encodable>(endpoint: String, body: T)
                            -> String {
        var responseString = ""
        let url = URL(string: baseURL + endpoint)!
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \(key)", forHTTPHeaderField: "Authorization")
        request.httpBody = try? JSONEncoder().encode(body)

        let semaphore = DispatchSemaphore(value: 0)
        URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                print("Error: \(error)")
            }
            if let data = data {
                responseString = String(data: data, encoding: .utf8) ?? "{}"
            }
            semaphore.signal()
        }.resume()
        semaphore.wait()

        return responseString
    }
```

```
    static func chat(messages: [[String: String]], maxTokens: Int = 25,
                                temperature: Double = 0.3) -> String {
        let chatRequest = ChatRequest(
            model: "gpt-4o-mini",
            messages: messages,
            max_tokens: maxTokens,
            temperature: temperature
        )

        let response = makeRequest(endpoint: "/chat/completions", body: chatRequest)
        guard let data = response.data(using: .utf8),
              let json = try? JSONSerialization.jsonObject(with: data)
                        as? [String: Any],
              let choices = json["choices"] as? [[String: Any]],
              let firstChoice = choices.first,
              let message = firstChoice["message"] as? [String: Any],
              let content = message["content"] as? String else {
            return ""
        }
        return content
    }

    static func embeddings(text: String) -> [Float] {
        let embeddingRequest = EmbeddingRequest(
            model: "text-embedding-ada-002",
            input: text
        )

        let response = makeRequest(endpoint: "/embeddings", body: embeddingRequest)
        guard let data = response.data(using: .utf8),
              let json = try? JSONSerialization.jsonObject(with: data)
                        as? [String: Any],
              let dataArray = json["data"] as? [[String: Any]],
              let embedding = dataArray.first?["embedding"] as? [NSNumber] else {
            return [1.23]
        }
        return embedding.map { number in Float(truncating: number) }
    }
}

// Usage functions:
func summarize(text: String, maxTokens: Int = 40) -> String {
    OpenAI.chat(messages: [
```

```
        ["role": "system",
         "content":
           "You are a helpful assistant that summarizes text concisely."],
        ["role": "user", "content": text]
    ], maxTokens: maxTokens)
}

func questionAnswering(question: String) -> String {
    OpenAI.chat(messages: [
        ["role": "system",
         "content":
           "You are a helpful assistant that answers questions directly and concisel\
y."],
        ["role": "user", "content": question]
    ], maxTokens: 25)
}

func completions(promptText: String, maxTokens: Int = 25) -> String {
    OpenAI.chat(messages: [["role": "user", "content": promptText]],
                          maxTokens: maxTokens)
}
```

This Swift implementation provides a streamlined interface to OpenAI's API services, focusing primarily on chat completions and text embeddings functionality. The code is structured around a central OpenAI struct that encapsulates all API interactions and provides a clean, type-safe interface for making requests.

Core Architecture

The implementation follows a modular design pattern, separating concerns between network communication, request/response handling, and utility functions. It utilizes Swift's strong type system through dedicated request models and leverages environment variables for secure API key management.

Key Features

Authentication and Configuration

The client automatically retrieves the OpenAI API key from environment variables, providing a secure way to handle authentication credentials. The base URL is configured as a constant, making it easy to modify for different environments or API versions.

Chat Completions

The chat completion functionality supports the GPT-4 model family, allowing for structured conversations through an array of messages. Each message contains a role (system, user, or assistant) and content. The implementation provides fine-grained control over:

- Maximum token output
- Temperature settings for response randomness
- Message context management
- Text embeddings

The embeddings feature implements OpenAI's text-embedding-ada-002 model, converting text inputs into high-dimensional vector representations. These embeddings can be used for:

- Semantic search
- Text similarity comparisons
- Document classification
- Other natural language processing tasks

Utility Functions

The implementation includes pre-built utility functions for common use cases:

- Text summarization with customizable length
- Question-answering with concise responses
- General text completions

Technical Implementation Details

Network Communication

The networking layer uses URLSession with a synchronous approach via DispatchSemaphore. While this ensures straightforward usage, it's worth noting that this approach should be carefully considered for production environments where asynchronous communication might be more appropriate.

Error Handling

The implementation includes basic error handling through Swift's optional binding and guard statements, providing graceful fallbacks for common failure scenarios. The embedding function, for instance, returns a default value rather than throwing an error when processing fails.

Data Parsing

JSON parsing is handled through a combination of JSONEncoder for requests and JSONSerialization for responses, with careful optional chaining to safely handle malformed or unexpected responses.

Running Tests

The file **SWIFT_BOOK/OpenAI_swift/Tests/OpenAI_swiftTests/OpenAI_swiftTests.swift** contains test code:

```
    import XCTest
    @testable import OpenAI_swift

    final class OpenAI_swiftTests: XCTestCase {
        func testExample() {
            print("Starting tests...")
            let embeds = OpenAI.embeddings(text: "Congress passed tax laws.")
            print(embeds[..<min(10, embeds.count)])
            let prompt = "He walked to the river and looked at"
            let ret = completions(promptText: prompt)
            print("** ret from OpenAI API call:", ret)
            let question = "Where was Leonardo da Vinci born?"
            let answer = questionAnswering(question: question)
            print("** answer from OpenAI API call:", answer)
            let text = "Jupiter is the fifth planet from the Sun and the largest in \
the Solar System. It is a gas giant with a mass one-thousandth that of the Sun, but \
two-and-a-half times that of all the other planets in the Solar System combined. Jup\
iter is one of the brightest objects visible to the naked eye in the night sky, and \
has been known to ancient civilizations since before recorded history. It is named a\
fter the Roman god Jupiter.[19] When viewed from Earth, Jupiter can be bright enough\
 for its reflected light to cast visible shadows,[20] and is on average the third-br\
ightest natural object in the night sky after the Moon and Venus."
            let summary = summarize(text: text)
            print("** generated summary: ", summary)
        }
    }
```

Output from this test code is:

```
$ swift test
Building for debugging...
[4/4] Compiling OpenAI_swift OpenAI_swift.swift
Build complete! (0.59s)
Test Suite 'All tests' started at 2024-11-17 16:42:12.354.
Test Suite 'OpenAI_swiftPackageTests.xctest' started at 2024-11-17 16:42:12.355.
Test Suite 'OpenAI_swiftTests' started at 2024-11-17 16:42:12.355.
Test Case '-[OpenAI_swiftTests.OpenAI_swiftTests testExample]' started.
Test Case '-[OpenAI_swiftTests.OpenAI_swiftTests testExample]' passed (7.429 seconds\
).
Test Suite 'OpenAI_swiftTests' passed at 2024-11-17 16:42:19.784.
         Executed 1 test, with 0 failures (0 unexpected) in 7.429 (7.429) seconds
Test Suite 'OpenAI_swiftPackageTests.xctest' passed at 2024-11-17 16:42:19.785.
         Executed 1 test, with 0 failures (0 unexpected) in 7.429 (7.430) seconds
Test Suite 'All tests' passed at 2024-11-17 16:42:19.785.
         Executed 1 test, with 0 failures (0 unexpected) in 7.429 (7.431) seconds
Starting tests...
[-0.0046315026, -0.0077434415, 0.0005571732, -0.024781546, -0.0031119392, -0.0185893\
25, -0.003362034, -0.020906659, 0.0074585234, -0.019463073]
** ret from OpenAI API call: the shimmering surface of the water, where the sunlight\
 danced in tiny sparkles. The gentle flow of the river whispered secrets as
** answer from OpenAI API call: Leonardo da Vinci was born in Vinci, Italy, on April\
 15, 1452.
** generated summary:  Jupiter is the fifth planet from the Sun and the largest in t\
he Solar System, being a gas giant with a mass one-thousandth that of the Sun and tw\
o-and-a-half times that of
* Test run started.
* Testing Library Version: 102 (arm64e-apple-macos13.0)
* Test run with 0 tests passed after 0.001 seconds.
```

Using APIs for Anthropic Claude LLMs

Here, I decided to not write a new client library for the Anthropic APIs since there are several existing high quality libraries for accessing the Anthropic Claude APIs.

This is not a strong recommendation of one Anthropic client library over another, but I very much enjoy using the following project because of the simplicity of its API:

```
https://github.com/fumito-ito/AnthropicSwiftSDK
```

My examples using this library to access the Anthropic Claude APIs can be found here:

```
https://github.com/mark-watson/Anthropic_swift_examples
```

You need to set the following environment variable for your person Anthropic API key: Anthropic API key:

```
ANTHROPIC_API_KEY
```

that you can get by creating an account:

```
https://console.anthropic.com
```

Note that there is no library implemented in this chapter.

Running the examples

All of the examples are packaged as Swift tests so git clone my examples repository https://github.com/mark-watson/Anthropic_swift_examples[31] and run:

```
swift test
```

The test Swift source file defines a test class (just the first few lines shown here):

[31] https://github.com/mark-watson/Anthropic_swift_examples

```
final class Anthropic_swift_examplesTests: XCTestCase {
  let text1 = "If Mary is 42, Bill is 27, and Sam is 51, what are their pairwise age\
 differences. Please be concise."
  func testExample() async throws {
    let key =
      ProcessInfo.processInfo.environment["ANTHROPIC_API_KEY"]!
    let anthropic = Anthropic(apiKey: key)
    let message = Message(role: .user,
                          content: [.text(text1)])
    let response =
      try await
        anthropic.messages.createMessage([message],
                                         maxTokens: 400)
```

If you print the value of **response** you see:

```
MessagesResponse(id: "msg_02RvimFjHFFmaV4n994J9wck", type: AnthropicSwiftSDK.Message\
sResponseType.message, role: AnthropicSwiftSDK.Role.assistant, content: [AnthropicSw\
iftSDK.Content.text("The pairwise age differences are:\n\nMary and Bill: 15 years\nM\
ary and Sam: 9 years\nBill and Sam: 24 years", cacheControl: nil)], model: Optional(\
AnthropicSwiftSDK.Model.claude_3_Opus), stopReason: Optional(AnthropicSwiftSDK.StopR\
eason.endTurn), stopSequence: nil, usage: AnthropicSwiftSDK.TokenUsage(inputTokens: \
Optional(38), outputTokens: Optional(38)))
```

If you print the value of **response.content** you see:

```
[AnthropicSwiftSDK.Content.text("The pairwise age differences are:\n\nMary and Bill:\
 15 years\nMary and Sam: 9 years\nBill and Sam: 24 years", cacheControl: nil)]
```

For normal use you want just the string contents of the model's response to your prompt, so use:

```
  for content in response.content {
    if case let .text(text, _) = content {
        print("Assistant's response: \(text)")
    }
  }
```

That outputs:

```
Assistant's response: The pairwise age differences are:

Mary and Bill: 15 years
Mary and Sam: 9 years
Bill and Sam: 24 years
```

In the general case of the Claude model returning images, tools used, and tool results, use code like this:

```
  for content in response.content {
     switch content {
       case .text(let text, _):
         print("Assistant's response: \(text)")
       case .image(let imageContent, _):
         print("imageContent: \(imageContent)")
         break
       case .document(let documentContent, _):
         print("documentContent: \(documentContent)")
         break
       case .toolResult(let toolResult):
         print("toolResult: \(toolResult)")
         break
       case .toolUse(let toolUse):
         print("toolUse: \(toolUse)")
         break
       }
     }
  }
```

Using Groq APIs to Open Weight LLM Models

Groq develops custom silicon for fast LLM inference.

Groq's API service supports a variety of openly available models, including:

- Llama 3.1 Series: Models like llama-3.1-70b-versatile, llama-3.1-8b-instant, and others, offering up to 128K context windows.
- Llama 3.2 Vision Series: Multimodal models such as llama-3.2-90b-vision-preview and llama-3.2-11b-vision-preview, capable of processing both text and image inputs.
- Llama 3 Groq Tool Use Models: Specialized for function calling, including llama3-Groq-70b-8192-tool-use-preview and llama3-Groq-8b-8192-tool-use-preview.
- Mixtral 8x7b: A model with a 32,768-token context window, suitable for extensive context applications.
- Gemma Series: Models like gemma2-9b-it and gemma-7b-it, each with an 8,192-token context window.
- Whisper Series: Models such as whisper-large-v3 and whisper-large-v3-turbo, designed for audio transcription and translation tasks.

To obtain an API key, visit Groq's API keys management page:

```
https://console.groq.com/keys
```

The code for this chapter can be found here:

```
https://github.com/mark-watson/Groq_swift
```

Implementation of a Client Library for the Groq APIs

Groq supports the OpenAI APIs so the following client library for Groq is similar to what I wrote previously for OpenAI:

```
import Foundation

struct Groq {
    private static let key = ProcessInfo.processInfo.environment["GROQ_API_KEY"]!
    private static let baseURL = "https://api.groq.com/openai/v1/"

    private static let MODEL = "llama3-8b-8192"

    private struct ChatRequest: Encodable {
        let model: String
        let messages: [[String: String]]
        let max_tokens: Int
        let temperature: Double
    }

    private static func makeRequest<T: Encodable>(endpoint: String, body: T)
                        -> String {
        var responseString = ""
        let url = URL(string: baseURL + endpoint)!
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \(key)", forHTTPHeaderField: "Authorization")
        request.httpBody = try? JSONEncoder().encode(body)

        let semaphore = DispatchSemaphore(value: 0)
        URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                print("Error: \(error)")
            }
            if let data = data {
                responseString = String(data: data, encoding: .utf8) ?? "{}"
            }
            semaphore.signal()
        }.resume()
        semaphore.wait()

        return responseString
    }

    static func chat(messages: [[String: String]],
                                maxTokens: Int = 25,
                                temperature: Double = 0.3)
```

```
                                                -> String {
        let chatRequest = ChatRequest(
            model: MODEL,
            messages: messages,
            max_tokens: maxTokens,
            temperature: temperature
        )

        let response = makeRequest(endpoint: "/chat/completions", body: chatRequest)
        guard let data = response.data(using: .utf8),
              let json = try? JSONSerialization.jsonObject(with: data)
                        as? [String: Any],
              let choices = json["choices"] as? [[String: Any]],
              let firstChoice = choices.first,
              let message = firstChoice["message"]
                                as? [String: Any],
              let content = message["content"] as? String else {
            return ""
        }
        return content
    }

}

// Usage functions:
func summarize(text: String, maxTokens: Int = 40) -> String {
    Groq.chat(messages: [
      ["role": "system",
       "content":
       "You are a helpful assistant that summarizes text concisely"],
      ["role": "user", "content": text]
    ], maxTokens: maxTokens)
}

func questionAnswering(question: String) -> String {
    Groq.chat(messages: [
      ["role": "system",
       "content":
       "You are a helpful assistant that answers questions directly and concisely."],
      ["role": "user", "content": question]
    ], maxTokens: 25)
}
```

```
func completions(promptText: String, maxTokens: Int = 25) -> String {
    Groq.chat(messages: [
        ["role": "system", "content": "You complete text"],
        ["role": "user", "content": promptText]],
        maxTokens: maxTokens)
}
```

Explanation of the Swift Groq API Code

1. Setting Up the API

- The `Groq` struct is designed to interact with the Groq API, mimicking OpenAI's API.
- **API Key**: Retrieved from the environment variable `GROQ_API_KEY`.
- **Base URL**: The API's base endpoint is `https://api.groq.com/openai/v1/`.
- **Model**: The model being used is predefined as `llama3-8b-8192`.

2. Structure of a Chat Request

- A private struct, `ChatRequest`, defines the JSON payload for requests:
  - `model`: The model name.
  - `messages`: A history of the conversation.
  - `max_tokens`: Limits the number of tokens in the response.
  - `temperature`: Controls randomness in the responses.

---

3. Making an HTTP POST Request

- The `makeRequest` function handles API communication:
  - Constructs the full URL by appending the endpoint to the base URL.
  - Sets up a POST request with:
    * JSON content type.
    * Authorization header using the API key.
  - Encodes the request body into JSON using `JSONEncoder`.
  - Sends the request asynchronously but waits for the response using a semaphore.
  - Parses the response data into a string.

---

4. Chat Functionality

- The `chat` function simplifies sending messages to the API:
  - Constructs a `ChatRequest` object with the given parameters.
  - Sends the request to the `/chat/completions` endpoint.
  - Processes the JSON response to extract the model's reply from the `choices` array.

---

5. Usage Functions

summarize - Summarizes a text. - Sends a conversation history where the system is described as "a helpful assistant that summarizes text concisely."

```
summarize(text: String, maxTokens: Int = 40)
```

questionAnswering - Answers a user-provided question directly. - Sends a conversation history where the system is described as "a helpful assistant that answers questions directly and concisely."

completions - Generates continuations for a given user prompt.

Running the Tests

Here is the test/example code for this library:

```
import XCTest
@testable import Groq_swift

final class Groq_swiftTests: XCTestCase {
  func testExample() {
    print("Starting tests...")
    let prompt = "He walked to the river and looked at"
    let ret = completions(promptText: prompt, maxTokens: 200)
    print("** ret from Groq API call:", ret)
    let question = "Where was Leonardo da Vinci born?"
    let answer = questionAnswering(question: question)
    print("** answer from Groq API call:", answer)
    let text = "Jupiter is the fifth planet from the Sun and the largest in the Sola\
r System. It is a gas giant with a mass one-thousandth that of the Sun, but two-and-\
a-half times that of all the other planets in the Solar System combined. Jupiter is \
one of the brightest objects visible to the naked eye in the night sky, and has been\
 known to ancient civilizations since before recorded history. It is named after the\
```

```
 Roman god Jupiter.[19] When viewed from Earth, Jupiter can be bright enough for its\
 reflected light to cast visible shadows,[20] and is on average the third-brightest \
natural object in the night sky after the Moon and Venus."
    let summary = summarize(text: text)
    print("** generated summary: ", summary)
  }
}
```

Here is sample output from this example use of the library:

```
Starting tests...

** ret from Groq API call: the calm water, feeling the warm sun on his face and the \
gentle breeze rustling his hair. The sound of the water lapping against the shore wa\
s soothing, and he closed his eyes, taking a deep breath to clear his mind.

** answer from Groq API call: Leonardo da Vinci was born on April 15, 1452, in Vinci\
, Italy.

** generated summary:  Here's a concise summary:

Jupiter is the largest planet in our Solar System, a gas giant with a mass 2.5 times\
 that of all other planets combined. It's the fifth planet
```

Using the xAI Grok LLM

xAI's Grok is a large language model (LLM) developed by Elon Musk's AI startup, xAI, to compete with leading AI systems like OpenAI's GPT-4. Launched in 2023, Grok is designed to handle a variety of tasks, including answering questions, assisting with writing, and solving coding problems. It is integrated with the social media platform X (formerly Twitter), providing users with real-time information and a conversational AI experience.

Grok has undergone several iterations, with Grok-2 being released in August 2024. This version introduced image generation capabilities, enhancing its versatility. xAI has also made Grok-1 open-source, allowing developers to access its weights and architecture for further research and application development.

To support Grok's development, xAI has invested in substantial computational resources, including the Colossus supercomputer, which utilizes 100,000 Nvidia H100 GPUs, positioning it as one of the most powerful AI training systems globally.

Implementation of a Grok API Client Library

The code for my xAI Grok client code can be found here:

```
https://github.com/mark-watson/X_GROK_swift
```

The Grok is similar to the OpenAI APIs so I copied the code we saw earlier that I wrote for OpenAI and made the simple modifications required to access Grok APIs (code discussion appears after this listing):

```
import Foundation

// xAI Grog LLM client library

struct X_GROK {
    private static let key = ProcessInfo.processInfo.environment["X_GROK_API_KEY"]!
    private static let baseURL = "https://api.x.ai/v1"

    private static let MODEL = "grok-beta"

    private struct ChatRequest: Encodable {
        let model: String
```

```
        let messages: [[String: String]]
        let max_tokens: Int
        let temperature: Double
    }

    private static func makeRequest<T: Encodable>(endpoint: String, body: T)
                                                -> String {
        var responseString = ""
        let url = URL(string: baseURL + endpoint)!
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \(key)", forHTTPHeaderField: "Authorization")
        request.httpBody = try? JSONEncoder().encode(body)

        let semaphore = DispatchSemaphore(value: 0)
        URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                print("Error: \(error)")
            }
            if let data = data {
                responseString = String(data: data, encoding: .utf8) ?? "{}"
            }
            semaphore.signal()
        }.resume()
        semaphore.wait()

        return responseString
    }

    static func chat(messages: [[String: String]], maxTokens: Int = 25,
                               temperature: Double = 0.3) -> String {
        let chatRequest = ChatRequest(
            model: MODEL,
            messages: messages,
            max_tokens: maxTokens,
            temperature: temperature
        )

        let response = makeRequest(endpoint: "/chat/completions",
                                   body: chatRequest)
        guard let data = response.data(using: .utf8),
              let json = try? JSONSerialization.jsonObject(with: data) as? [String: \
```

```
Any],
              let choices = json["choices"] as? [[String: Any]],
              let firstChoice = choices.first,
              let message = firstChoice["message"]
                                as? [String: Any],
              let content = message["content"] as? String else {
            return ""
        }
        return content
    }
}

// Usage functions:
func summarize(text: String, maxTokens: Int = 40) -> String {
    X_GROK.chat(messages: [
      ["role": "system",
       "content":
       "You are a helpful assistant that summarizes text concisely."],
      ["role": "user", "content": text]
    ], maxTokens: maxTokens)
}

func questionAnswering(question: String) -> String {
    X_GROK.chat(messages: [
      ["role": "system",
       "content":
       "You are a helpful assistant who answers questions directly and concisely."],
      ["role": "user", "content": question]
    ], maxTokens: 25)
}

func completions(promptText: String, maxTokens: Int = 25) -> String {
    X_GROK.chat(messages: [["role": "user", "content": promptText]],
                          maxTokens: maxTokens)
}
```

This Swift code defines a client library, **X_GROK**, to interact with the xAI Grok Large Language Model (LLM) API. It leverages Swift's Foundation framework to handle HTTP requests and JSON encoding/decoding. The library retrieves the API key from the environment variable X_GROK_-API_KEY and sets the base URL for the API. It specifies a default model, grok-beta, for generating responses.

The core functionality is encapsulated in the makeRequest function, which constructs and sends HTTP POST requests to the API. It accepts an endpoint and a request body conforming to the

Encodable protocol. The function sets the necessary HTTP headers, including Content-Type and Authorization, and encodes the request body into JSON. To handle the asynchronous nature of network calls synchronously, it employs a semaphore, ensuring the function waits for the response before proceeding. The response is then returned as a string.

The chat function utilizes makeRequest to send chat messages to the API. It constructs a ChatRequest struct with parameters like the model, messages, maximum tokens, and temperature. After receiving the response, it parses the JSON to extract the generated content. Additionally, the code provides utility functions **summarize**, **questionAnswering**, and **completions** which use the **chat** function to perform specific tasks such as text summarization, question answering, and text completion, respectively.

Here is the test/example code for this library:

```
import XCTest
@testable import X_GROK_swift

final class X_GROK_swiftTests: XCTestCase {
  func testExample() {
    print("Starting tests...")
    let prompt = "He walked to the river and looked at"
    let ret = completions(promptText: prompt, maxTokens: 200)
    print("** ret from X_GROK API call:", ret)
    let question = "Where was Leonardo da Vinci born?"
    let answer = questionAnswering(question: question)
    print("** answer from X_GROK API call:", answer)
    let text = "Jupiter is the fifth planet from the Sun and the largest in the Sola\
r System. It is a gas giant with a mass one-thousandth that of the Sun, but two-and-\
a-half times that of all the other planets in the Solar System combined. Jupiter is \
one of the brightest objects visible to the naked eye in the night sky, and has been\
 known to ancient civilizations since before recorded history. It is named after the\
 Roman god Jupiter.[19] When viewed from Earth, Jupiter can be bright enough for its\
 reflected light to cast visible shadows,[20] and is on average the third-brightest \
natural object in the night sky after the Moon and Venus."
    let summary = summarize(text: text)
       print("** generated summary: ", summary)
    }
}
```

Here is the example code output:

```
Starting tests...
** ret from X_GROK API call: He walked to the river and looked at the water flowing \
gently by. The serene scene provided a moment of peace, as he watched the ripples da\
nce in the sunlight. The sound of the water was soothing, almost like a lullaby, cal\
ming his thoughts and grounding him in the present. He felt a deep connection to nat\
ure, the river's timeless flow reminding him of life's continuous journey.
** answer from X_GROK API call: Leonardo da Vinci was born in the town of Vinci, in \
the region of Tuscany, Italy.
** generated summary:  Jupiter, the fifth planet from the Sun, is the largest in our\
 Solar System, classified as a gas giant with a mass significantly greater than all \
other planets combined. It's one of the brightest objects
```

Using Ollama to Run Local LLMs

Ollama is a program and framework written in Go that allows you to download, run models on the command line, and call using a REST style interface. You need to downnload the Ollama executable for your operation system at https://ollama.com[32].

Similarly to our use of a third party for accessing the Anthropic Clause models, here we will not write a wrapper libary. The example code for ths chapter is in the test code for the Swift project in the GitHub repository https://github.com/mark-watson/Ollama_swift_examples[33].

We use the library in the GitHub repository https://github.com/mattt/ollama-swift[34].

Running the Ollama Service

Assuming you have Ollama installed, download the following model that required two gigabytes of disk space:

```
ollama pull llama3.2:latest
```

When the model is downloaded it is also cached for future use on your laptop.

Here is the test/example code we will run:

```
import XCTest
import Ollama

final class Ollama_swift_examplesTests: XCTestCase {
    let text1 = "If Mary is 42, Bill is 27, and Sam is 51, what are their pairwise a\
ge differences."
    let client = Ollama.Client.default // http://localhost:11434 endpoint
    func testExample() async throws {
      let response = try await client.chat(
        model: "llama3.2:latest",
        messages: [
            .system("You are a helpful assistant who completes text and also answers\
 questions. You are always concise."),
            .user(text1),
```

[32] https://ollama.com
[33] https://github.com/mark-watson/Ollama_swift_examples
[34] https://github.com/mattt/ollama-swift

```
            .user("what if Sam is 52?")
        ])
        print(response.message.content)
    }
}
```

The output looks like:

```
Pairwise age differences:

- Mary - Bill: |42 - 27| = 15
- Mary - Sam: |42 - 51| = 9
- Bill - Sam: |27 - 51| = 24

If Sam is 52:
- Mary - Bill: |42 - 27| = 15
- Mary - Sam: |42 - 52| = 10
- Bill - Sam: |27 - 52| = 25
```

The **ollama_swift** library also supports text generation. You can also do single shot text generation using the code in the previous example, but only using one **user** call, for example:

```
final class Ollama_swift_examplesTests: XCTestCase {
    let text1 = "What is the capital of Germany?"
    let client = Ollama.Client.default
    func testExample() async throws {
      let response = try await client.chat(
        model: "llama3.2:latest",
        messages: [
            .system("You are a helpful assistant who completes text and also answers\
 questions. You are always concise."),
            .user(text1),
        ])
        print(response.message.content)
    }
}
```

The output looks like:

```
The capital of Germany is Berlin.
```

Ollama Wrap Up

This is a short chapter but an important one. I do over half my work with LLMs running locally on my laptop using Ollama, with the rest of my work using OpenAI, Anthropic, and Groq commercial APIs.

Using Apple's MLX Framework to Run Local LLMs

Apple's MLX framework is an efficient way to use LLMs embedded in applications written in Swift using the SwiftUI user interface library for macOS, iOS, and iPadOS.

It is difficult to create simple command line Swift apps using MLX but there are several complete MLX, Swift, and SwiftUI demo applications that you can use to start your own projects. Here we will use the **LLMEval** application from the GitHub repository https://github.com/ml-explore/mlx-swift-examples[35].

MLX Framework History

Apple's MLX framework, introduced in December 2023, is a key part of Apple's strategy to support AI on its hardware platforms by leveraging the unique capabilities of Apple Silicon, including the M1, M2, M3, and M4 series. Designed as an open-source, NumPy-like array framework, MLX optimizes machine learning workloads, particularly large language models (LLMs), by utilizing Apple Silicon's unified architecture that integrates CPU, GPU, Neural Engine, and shared memory. This architecture eliminates data transfer bottlenecks, enabling faster and more efficient ML tasks, such as training and deploying LLMs directly on devices like MacBooks and iPhones. MLX aligns with Apple's privacy-focused approach by supporting on-device processing, enhancing performance for applications like natural language processing, speech recognition, and content generation while offering a seamless transition for Python or Swift ML engineers familiar with frameworks like NumPy and PyTorch. MLX stands out by leveraging Apple's unified memory architecture, allowing shared memory access between CPU and GPU, which eliminates data transfer overhead and accelerates machine learning tasks, especially with large datasets.

MLX Resources on GitHub

In this chapter we will look at an example application that is part of the Swift MLX Examples project. After working through this example, the following resources on GitHub are worth looking at:

- https://github.com/ml-explore/mlx-swift: The Swift API for MLX, enabling integration with Swift-based projects.

[35] https://github.com/ml-explore/mlx-swift-examples

- https://github.com/ml-explore/mlx-swift-examples: Examples showcasing the use of MLX with Swift.

You can find the documentation here:

https://swiftpackageindex.com/ml-explore/mlx-swift/0.18.0/documentation/mlx[36].

These repositories provide a comprehensive set of tools and examples to effectively utilize MLX for machine learning tasks on Apple silicon. There are many other repositories for MLX and Python and if you need to perform tasks like fine tuning a MLX model, that task should probably be done using Python.

Example Application for MLX Swift Examples Repository

You will want to download the complete MLX Swift examples repository:

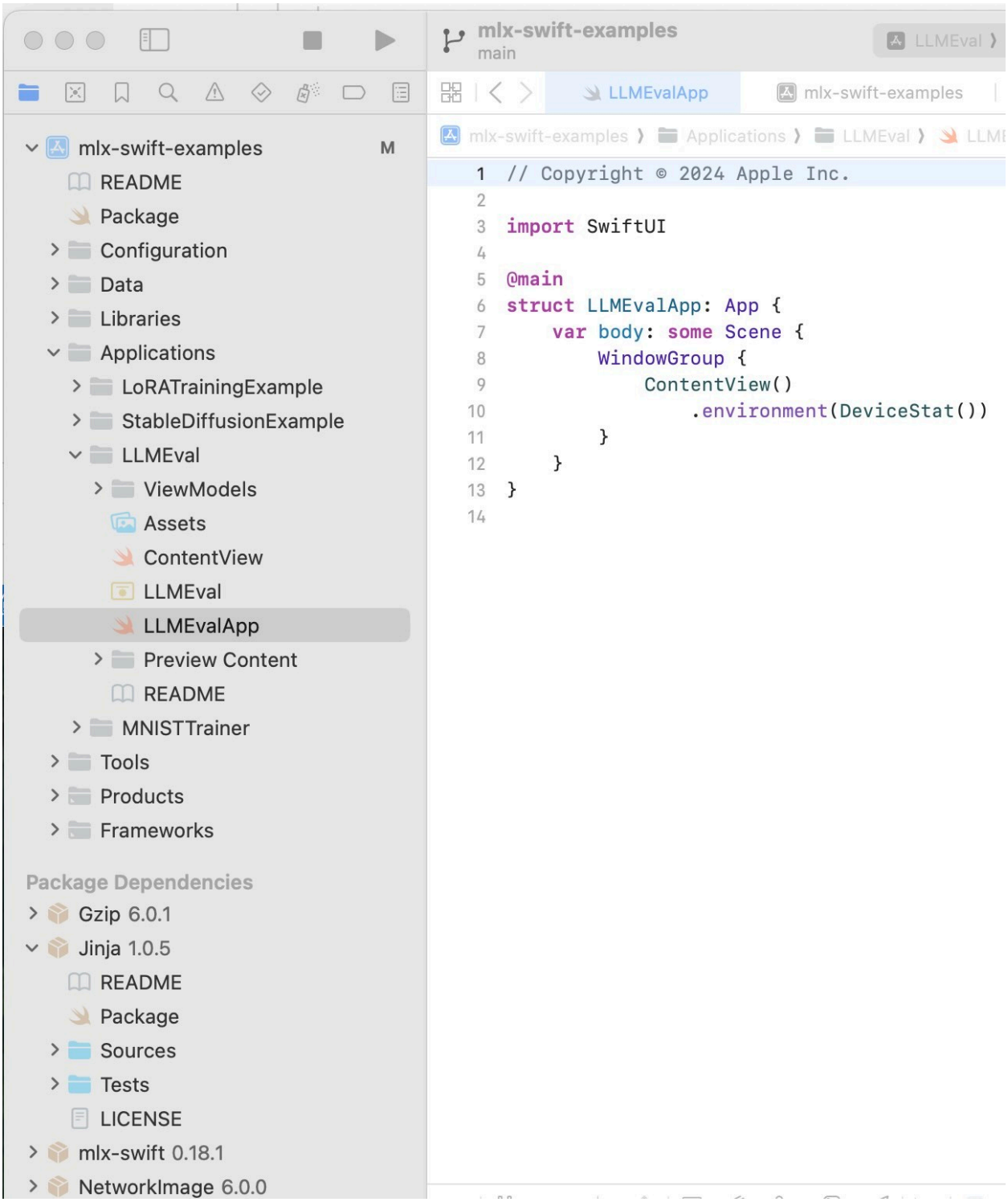
```
git clone https://github.com/ml-explore/mlx-swift-examples.git
```

Open the top level XCode project by:

```
cd mlx-swift-examples
open mlx-swift-examples.xcodeproj
```

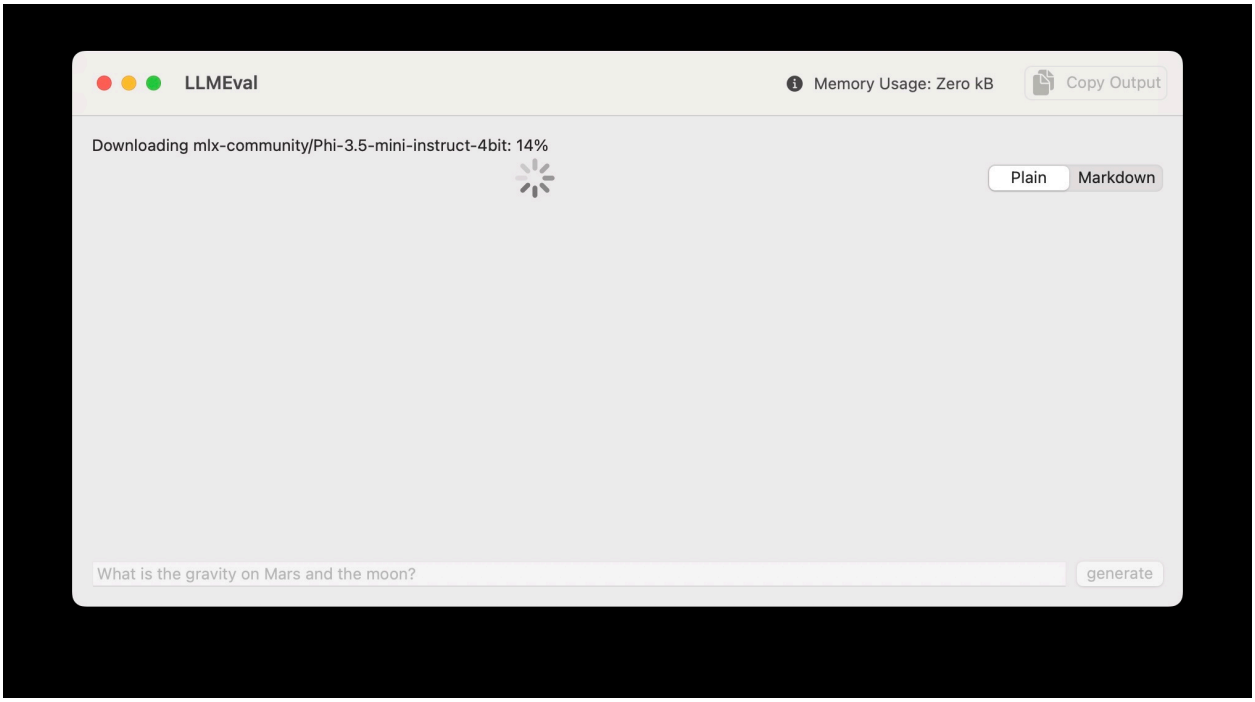
Here is the file browser view of this project:

[36] https://swiftpackageindex.com/ml-explore/mlx-swift/0.18.0/documentation/mlx



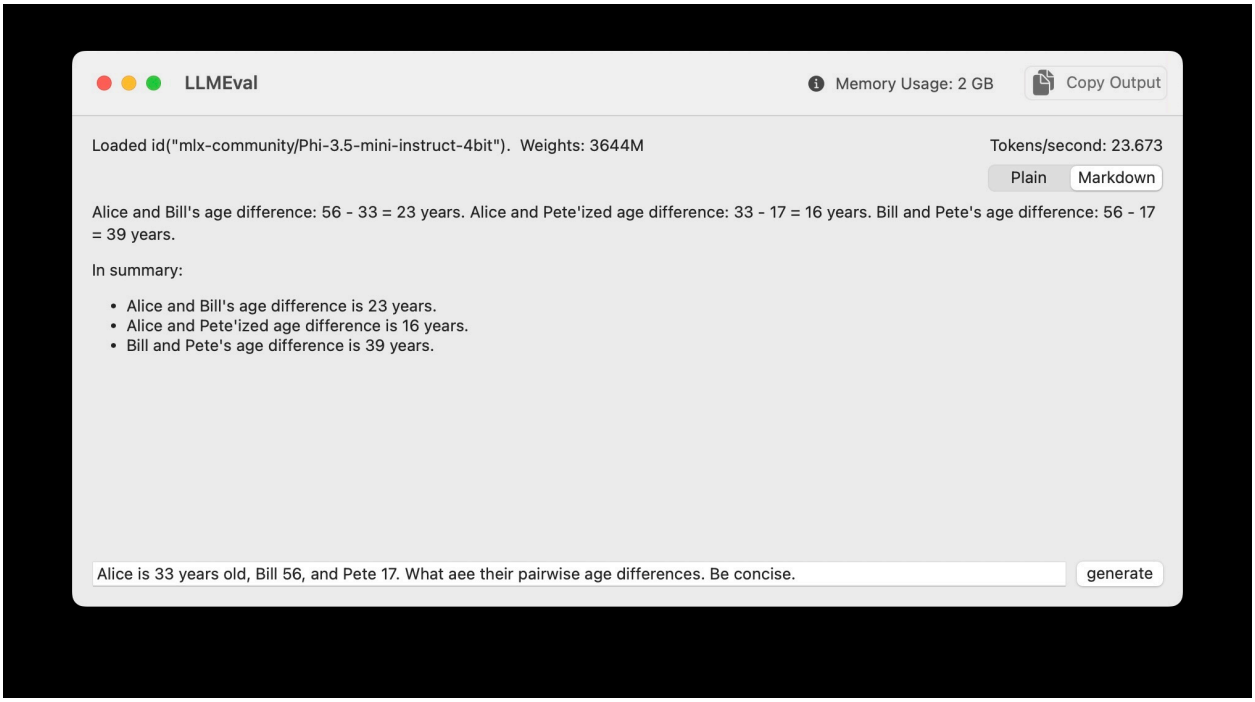
XCode View of projects

Running the LLMEval project:



LLMEval app downloading model file

Initially the model is downloaded and cached on your laptop for future use. Here is the app used to solve a simple word problem:



LLMEval app answering user's question

Analysis of Swift and SwiftUI Code in the LLMEval Application

This example is part of the Swift MLX Examples project that currently has twenty contributors and a thousand stars on GitHub https://github.com/ml-explore/mlx-swift-examples[37].

Unfortunately the SwiftUI user interface code is mixed in with the code that uses MLX. Let's walk through the code:

Here is a walk through a Swift-based program using Apple's frameworks for Machine Learning and Language Models with the code interspersed with explanations.

Imports

```
import LLM
import MLX
import MLXRandom
import MarkdownUI
import Metal
import SwiftUI
import Tokenizers
```

These imports bring in essential libraries:

- LLM and MLX for working with language models.
- MarkdownUI for rendering Markdown content.
- SwiftUI for creating the user interface.
- Tokenizers for tokenizing text.

The ContentView Struct

The ContentView struct defines the main interface of the app.

State Variables

```
struct ContentView: View {

  @State var prompt = ""
  @State var llm = LLMEvaluator()
  @Environment(DeviceStat.self) private var deviceStat
```

[37] https://github.com/ml-explore/mlx-swift-examples

In this code snippet:

- @State allows the view to track changes in the prompt and llm instances.
- @Environment fetches device statistics, such as GPU memory usage.

Display Style Enum

```
enum displayStyle: String, CaseIterable, Identifiable {
  case plain, markdown
  var id: Self { self }
}

@State private var selectedDisplayStyle = displayStyle.markdown
```

In this code snippet:

- **displayStyle** defines whether the output is plain text or Markdown.
- A segmented picker toggles between the two styles.

UI Layout

Input Section

```
var body: some View {
  VStack(alignment: .leading) {
    VStack {
     HStack {
       Text(llm.modelInfo).textFieldStyle(.roundedBorder)
       Spacer()
       Text(llm.stat)
     }
     HStack {
       Spacer()
       if llm.running {
          ProgressView().frame(maxHeight: 20)
          Spacer()
        }
        Picker("", selection: $selectedDisplayStyle) {
          ForEach(displayStyle.allCases, id: \.self) {
            option in
              Text(option.rawValue.capitalized)
```

```
                    .tag(option)
            }

        }.pickerStyle(.segmented)
    }
}
```

This code displays model information and statistics.

Output Section

```
ScrollView(.vertical) {
    ScrollViewReader { sp in
        Group {
            if selectedDisplayStyle == .plain {
                Text(llm.output)
                    .textSelection(.enabled)
            } else {
                Markdown(llm.output)
                    .textSelection(.enabled)
            }
        }
        .onChange(of: llm.output) { _, _ in
            sp.scrollTo("bottom")
        }
    }
}

HStack {
    TextField("prompt", text: $prompt)
                .onSubmit(generate)
                .disabled(llm.running)
    Button("generate", action: generate)
                .disabled(llm.running)
 }
}
```

The ScrollView shows the model's output, which updates dynamically as the model generates text.

Toolbar

```
      .toolbar {
         ToolbarItem {
           Label(
            "Memory Usage: \(deviceStat.gpuUsage.activeMemory.formatted(.byteCount(s\
tyle: .memory)))",
            systemImage: "info.circle.fill"
           )
          }
          ToolbarItem(placement: .primaryAction) {
             Button {
               Task {
                copyToClipboard(llm.output)
               }
              } label: {
               Label("Copy Output", systemImage: "doc.on.doc.fill")
              }
           }
        }
```

The toolbar includes:

- GPU memory usage information.
- A "Copy Output" button to copy the generated text.

The LLMEvaluator Class

This class handles the logic for loading and generating text with the language model.

Core Properties

```
@Observable
@MainActor
class LLMEvaluator {
  var running = false
  var output = ""
  var modelInfo = ""
  var stat = ""
  let modelConfiguration = ModelConfiguration.phi3_5_4bit
  /// parameters controlling the output
  let generateParameters = GenerateParameters(temperature: 0.6)
  let maxTokens = 240

  /// update the display every N tokens -- 4 looks like it updates continuously
```

```
  /// and is low overhead.  observed ~15% reduction in tokens/s when updating
  /// on every token
  let displayEveryNTokens = 4

  enum LoadState {
        case idle
        case loaded(ModelContainer)
  }

  var loadState = LoadState.idle
```

This code snippet:

- Tracks the model state and output.
- Configures the model (phi3_5_4bit).

Loading the Model (if required)

```
  /// load and return the model -- can be called
  /// multiple times, subsequent calls will
  /// just return the loaded model

  func load() async throws -> ModelContainer {
    switch loadState {
    case .idle:
        MLX.GPU.set(cacheLimit: 20 * 1024 * 1024)
        let modelContainer =
          try await LLM.loadModelContainer(configuration: modelConfiguration) {
            [modelConfiguration] progress in
            Task { @MainActor in
                self.modelInfo = "Downloading \(modelConfiguration.name): \(Int(prog\
ress.fractionCompleted * 100))%"
            }
        }
        self.modelInfo = "Loaded \(modelConfiguration.id). Weights: \(numParams / (1\
024*1024))M"
        loadState = .loaded(modelContainer)
        return modelContainer
    case .loaded(let modelContainer):
        return modelContainer
    }
  }
```

This code snippet:

- Downloads and caches the model.
- Updates modelInfo during the download.

Generating Output

```
func generate(prompt: String) async {
  guard !running else { return }
  running = true
  self.output = ""

  do {
      let modelContainer = try await load()
      let messages = [["role": "user", "content": prompt]]
      let promptTokens = try await modelContainer.perform { _, tokenizer in
          try tokenizer.applyChatTemplate(messages: messages)
      }

      let result = await modelContainer.perform { model, tokenizer in
          LLM.generate(
              promptTokens: promptTokens,
              parameters: generateParameters, model: model,
              tokenizer: tokenizer,
              extraEOSTokens: modelConfiguration.extraEOSTokens
          ) { tokens in
              if tokens.count % displayEveryNTokens == 0 {
                  let text = tokenizer.decode(tokens: tokens)
                  Task { @MainActor in
                      self.output = text
                  }
              }
              if tokens.count >= maxTokens {
                  return .stop
              } else {
                  return .more
              }
          }
      }
  } catch {
      output = "Failed: \(error)"
  }
```

```
    running = false
  }
}
```

This code snippet:

- Prepares the prompt for the model.
- Generates tokens and dynamically updates the view.

This program demonstrates how to integrate ML and UI components for interactive LLM-based applications in Swift.

This code example uses the MIT License so you can modify the example code if you need to write a combined SwiftUI GUI app that uses LLM-based text generation.

Part 3: Apple's CoreML and NLP Libraries

In this part we cover:

- Short introduction to the ideas behind Deep Learning
- Introduction of CoreML
- Examples using CoreML
- Introduction of NLP
- Examples using NLP libraries

This section used to contain Apple CoreML examples to train a back-propagation model from the University of Wisconsin cancer data set. As of April 2022, these example do not work because of a problem with latest CreateML library so this material has been removed from this book.

Deep Learning Introduction

Apple's work in smoothly integrating deep learning into their developer tools for macOS, iOS, and iPadOS applications is in my opinion nothing short of brilliant. We will finish this book with an application that uses two deep learning models that provide almost all of the functionality of the application.

Before diving into Apple's CoreML libraries in later chapters we will take a shallow dive into the principles of deep learning and take a lay-of-the-land look at the type of most commonly used models. This chapter has no example programs and is intended as background material.

Most of my professional career since 2014 has involved Deep Learning, mostly with TensorFlow using the Keras APIs. In the late 1980s I was on a DARPA neural network technology advisory panel for a year, I wrote the first prototype of the SAIC ANSim neural network library commercial product, and I wrote the neural network prediction code for a bomb detector my company designed and built for the FAA for deployment in airports. More recently I have used GAN (generative adversarial networks) models for synthesizing numeric spreadsheet data and LSTM (long short term memory) models to synthesize highly structured text data like nested JSON and for NLP (natural language processing). I have also written a product recommendation model for an online store using TensorFlow Recommenders. I have several USA and European patents using neural network and Deep Learning technology.

Here we will learn a vocabulary for discussing Deep Learning neural network models and look at possible architectures.

If you want to use Deep Learning professionally, there are two specific online resources that I recommend: Andrew Ng leads the efforts at deeplearning.ai[38] and Jeremy Howard leads the efforts at fast.ai[39].

There are many Deep Learning neural architectures in current practical use; a few types that I use are:

- Multi-layer perceptron networks with many fully connected layers. An input layer contains placeholders for input data. Each element in the input layer is connected by a two-dimensional weight matrix to each element in the first hidden layer. We can use any number of fully connected hidden layers, with the last hidden layer connected to an output layer.
- Convolutional networks for image processing and text classification. Convolutions, or filters, are small windows that can process input images (filters are two-dimensional) or sequences like text (filters are one-dimensional). Each filter uses a single set of learned weights independent of where the filter is applied in an input image or input sequence.

[38] https://www.deeplearning.ai/
[39] https://www.fast.ai/

- Autoencoders have the same number of input layer and output layer elements with one or more hidden fully connected layers. Autoencoders are trained to produce the same output as training input values using a relatively small number of hidden layer elements. Autoencoders are capable of removing noise in input data.
- LSTM (long short term memory) process elements in a sequence in order and are capable of remembering patterns that they have seen earlier in the sequence.
- GAN (generative adversarial networks) models comprise two different and competing neural models, the generator and the discriminator. GANs are often trained on input images (although in my work I have applied GANs to two-dimensional numeric spreadsheet data). The generator model takes as input a "latent input vector" (this is just a vector of specific size with random values) and generates a random output image. The weights of the generator model are trained to produce random images that are similar to how training images look. The discriminator model is trained to recognize if an arbitrary output image is original training data or an image created by the generator model. The generator and discriminator models are trained together.

The core functionality of libraries like TensorFlow are written in C++ and take advantage of special hardware like GPUs, custom ASICs, and devices like Google's TPUs. Most people who work with Deep Learning models don't need to even be aware of the low level optimizations used to make training and using Deep Learning models more efficient. That said, in the following section I am going to show you how simple neural networks are trained and used.

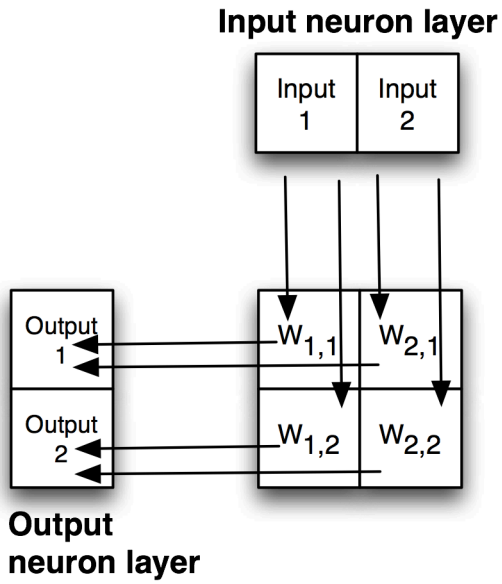
Simple Multi-layer Perceptron Neural Networks

I use the terms Multi-layer perceptron neural networks, backpropagation neural networks and delta-rule networks interchangeably. Backpropagation refers to the model training process of calculating the output errors when training inputs are passed in the forward direction from input layer, to hidden layers, and then to the output layer. There will be an error which is the difference between the calculated outputs and the training outputs. This error can be used to adjust the weights from the last hidden layer to the output layer to reduce the error. The error is then backprogated backwards through the hidden layers, updating all weights in the model. I have detailed example code in several of my older artificial intelligence books. Here I am satisfied to give you an intuition of how simple neural networks are trained.

The basic idea is that we start with a network initialized with random weights and for each training case we propagate the inputs through the network towards the output neurons, calculate the output errors, and back-up the errors from the output neurons back towards the input neurons in order to make small changes to the weights to lower the error for the current training example. We repeat this process by cycling through the training examples many times.

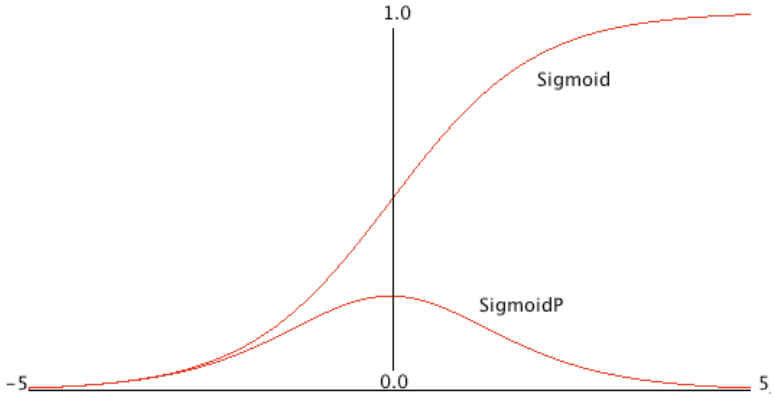
The following figure shows a simple backpropagation network with one hidden layer. Neurons in adjacent layers are connected by floating point connection strength weights. These weights start out as small random values that change as the network is trained. Weights are represented in the

following figure by arrows; in the code the weights connecting the input to the output neurons are represented as a two-dimensional array.



Example Backpropagation network with One Hidden Layer

Each non-input neuron has an activation value that is calculated from the activation values of connected neurons feeding into it, gated (adjusted) by the connection weights. For example, in the above figure, the value of Output 1 neuron is calculated by summing the activation of Input 1 times weight W1,1 and Input 2 activation times weight W2,1 and applying a "squashing function" like Sigmoid or Relu (see figures below) to this sum to get the final value for Output 1's activation value. We want to flatten activation values to a relatively small range but still maintain relative values. To do this flattening we use the Sigmoid function that is seen in the next figure, along with the derivative of the Sigmoid function which we will use in the code for training a network by adjusting the weights.



Sigmoid Function and Derivative of Sigmoid Function (SigmoidP)

Simple neural network architectures with just one or two hidden layers are easy to train using backpropagation and I have from scratch code (using no libraries) for this several of my previous books. However, here we are using Hy to write models using the TensorFlow framework which has the huge advantage that small models you experiment with on your laptop can be scaled to more parameters (usually this means more neurons in hidden layers which increases the number of weights in a model) and run in the cloud using multiple GPUs.

Except for pedantic purposes, I now never write neural network code from scratch. I take instead advantage of the many person-years of engineering work put into the development of frameworks like TensorFlow, PyTorch, mxnet, etc. We now move on to two examples built with TensorFlow.

Deep Learning

Deep Learning models are generally understood to have many more hidden layers than simple multi-layer perceptron neural networks and often comprise multiple simple models combined together in series or in parallel. Complex architectures can be iteratively developed by manually adjusting the size of model components, changing the components, etc. Alternatively, model architecture search can be automated. At Capital One I used Google's AdaNet project[40] that efficiently searches for effective model architectures inside a single TensorFlow session. Now all major cloud compute provides support some form of AutoML. You need to make a decision for yourself how much effort you want to put into deeply understanding the technology, or simply learning how to use pre-trained models.

[40] https://github.com/tensorflow/adanet

Natural Language Processing Using Apple's Natural Language Framework

I have been working in the field of Natural Language Processing (NLP) since 1985 so I 'lived through' the revolutionary change in NLP that has occurred since 2014: Deep Learning results out-classed results from previous symbolic methods.

https://developer.apple.com/documentation/naturallanguage

I will not cover older symbolic methods of NLP here, rather I refer you to my previous books Practical Artificial Intelligence Programming With Java[41], Loving Common Lisp, or the Savvy Programmer's Secret Weapon[42], and Haskell Tutorial and Cookbook[43] for examples. We get better results using Deep Learning (DL) for NLP and the libraries that Apple provides.

You will learn how to apply both DL and NLP by using the state-of-the-art full-feature libraries that Apple provides in their iOS and macOS development tools.

Using Apple's NaturalLanguage Swift Library

We will use one of Apple's NLP libraries consisting of pre-built models in the last chapter of this book. In order to fully understand the example in the last chapter you will need to read Apple's high-level discussion of using CoreML https://developer.apple.com/documentation/coreml[44] and their specific support for NLP https://developer.apple.com/documentation/naturallanguage/[45].

There are many pre-trained CoreML compatible models on the web, both from Apple and also from third party (e.g., https://github.com/likedan/Awesome-CoreML-Models[46]).

Apple also provides tools for converting TensorFlow and PyTorch models to be compatible with CoreML https://coremltools.readme.io/docs[47].

A simple Wrapper Library for Apple's NLP Models

I will not go into too much detail here but I created a small wrapper library for Apple's NLP models that will make it easier for you to jump in and have fun experimenting with them: https://github.com/mark-watson/Nlp_swift[48].

[41] https://leanpub.com/javaai
[42] https://leanpub.com/lovinglisp
[43] https://leanpub.com/haskell-cookbook
[44] https://developer.apple.com/documentation/coreml
[45] https://developer.apple.com/documentation/naturallanguage/
[46] https://github.com/likedan/Awesome-CoreML-Models
[47] https://coremltools.readme.io/docs
[48] https://github.com/mark-watson/Nlp_swift

The main library implementation file uses the **@available(OSX 10.13, *)** attribute to indicate that the following function is available on macOS 10.13 and later versions.

```
import Foundation
import NaturalLanguage

let tagger = NSLinguisticTagger(tagSchemes:[.tokenType, .language, .lexicalClass,
    .nameType, .lemma], options: 0)
let options: NSLinguisticTagger.Options = [.omitPunctuation, .omitWhitespace,
    .joinNames]

@available(OSX 10.13, *)
public func getEntities(for text: String) -> [(String, String)] {
    var words: [(String, String)] = []
    tagger.string = text
    let range = NSRange(location: 0, length: text.utf16.count)
    tagger.enumerateTags(in: range, unit: .word, scheme: .nameType,
    options: options) { tag, tokenRange, stop in
        let word = (text as NSString).substring(with: tokenRange)
        words.append((word, tag?.rawValue ?? "unkown"))
    }
    return words
}

@available(OSX 10.13, *)
public func getLemmas(for text: String) -> [(String, String)] {
    var words: [(String, String)] = []
    tagger.string = text
    let range = NSRange(location: 0, length: text.utf16.count)
    tagger.enumerateTags(in: range, unit: .word, scheme: .lemma,
            options: options) { tag, tokenRange, stop in
        let word = (text as NSString).substring(with: tokenRange)
        words.append((word, tag?.rawValue ?? "unkown"))
    }
    return words
}
```

The public function **getEntities** takes a String parameter called **text** and returns an array of tuples containing **(String, String)**. Here's a breakdown of what this function does:

- The function initializes an empty array called words to store the extracted entities.
- The line **tagger.string = text** sets the input text for a tagger object. The tagger is an instance of **NSLinguisticTagger**, which is a natural language processing class provided by Apple's Foundation framework.

- The next line creates an **NSRange** object called **range** that represents the entire length of the input text.
- The **tagger.enumerateTags(in:range, unit:.word, scheme:.nameType, options:options)** method is called to iterate over the words in the input text and extract their associated tags. The **in:** parameter specifies the range of the text to process. The **unit:** parameter specifies that the enumeration should be done on a word-by-word basis. The **scheme:** parameter specifies the linguistic scheme to use, in this case, the **.nameType** scheme, which is used to identify named entities. The **options:** parameter specifies additional options or settings for the tagger.
- Inside the enumeration block, the code retrieves the current word and its associated tag using the **tokenRange** and **tag** parameters.
- The line **let word = (text as NSString).substring(with: tokenRange)** extracts the substring corresponding to the current word using **tokenRange**.
- The line **words.append((word, tag?.rawValue ?? "unknown"))** appends a tuple containing the extracted word and its associated tag to the words array. If the tag is nil, it uses the default value of "unknown".
- Finally, the words array is returned, which contains all the extracted entities (words and their associated tags) from the input text.

The public function called **getLemmas** that takes a String parameter called **text** and returns an array of tuples containing **(String, String)**. Here's a breakdown of what the function **getLemmas** is very similar to the last function **getEntities**. The function **getLemmas** does the following:

- The function initializes an empty array called words to store the extracted lemmas.
- The line tagger.string = text sets the input text for a tagger object.
- The next line creates an **NSRange** object called **range** that represents the entire length of the input text.
- The **tagger.enumerateTags(in:range, unit:.word, scheme:.lemma, options: options)** method is called to iterate over the words in the input text and extract their corresponding lemmas.
- Inside the enumeration block, the code retrieves the current word and its associated lemma using the tokenRange and tag parameters.
- The line **let word = (text as NSString).substring(with: tokenRange)** extracts the substring corresponding to the current word using **tokenRange**.
- Finally, the words array is returned, which contains all the extracted lemmas (words and their associated base forms) from the input text.

In summary, function **getLemmas** uses the **NSLinguisticTagger** to perform linguistic analysis on a given text and extract the base forms (lemmas) of words. The lemmas are then stored in an array of tuples and returned as the result of the function.

Here is some test code:

```
let quote = "President George Bush went to Mexico with IBM representatives. Here's t\
o the crazy ones. The misfits. The rebels. The troublemakers. The round pegs in the \
square holes. The ones who see things differently. They're not fond of rules. And th\
ey have no respect for the status quo. You can quote them, disagree with them, glori\
fy or vilify them. About the only thing you can't do is ignore them. Because they ch\
ange things. They push the human race forward. And while some may see them as the cr\
azy ones, we see genius. Because the people who are crazy enough to think they can c\
hange the world, are the ones who do. - Steve Jobs (Founder of Apple Inc.)"
if #available(OSX 10.13, *) {
            print("\nEntities:\n")
            print(getEntities(for: quote))
            print("\nLemmas:\n")
            print(getLemmas(for: quote))
}
```

Here is an edited listing of the output with most of the output removed for brevity:

```
Entities:

[("President", "OtherWord"), ("George Bush", "PersonalName"), ("went", "OtherWord"),\
 ("to", "OtherWord"), ("Mexico", "PlaceName"), ("with", "OtherWord"), ("IBM", "Organ\
izationName"),
  ...]

Lemmas:

[("President", "President"), ("George Bush", "George"), ("went", "go"), ("to", "to")\
, ("Mexico", "Mexico"),
  ...]
```

Documents Question Answering Using OpenAI GPT4 APIs and a Local Embeddings Vector Database

The examples in this chapter are inspired by the Python LangChain and LlamaIndex projects, with just the parts I need for my projects written from scratch in Common Lisp. I wrote a Python book "LangChain and LlamaIndex Projects Lab Book: Hooking Large Language Models Up to the Real World Using GPT-3, ChatGPT, and Hugging Face Models in Applications" in March 2023: https://leanpub.com/langchain that you might also be interested in.

The GitHub repository for this example can be found here: https://github.com/mark-watson/Docs_-QA_Swift[49].

The entire example is in one Swift source file **main.swift**. All of the program listings in this chapter can be found in this single source file.

We use two models in this example: a vector embedding model and a gpt-4o-mini conversation model (see bottom of this file). The vector embedding model is used to generate a vector embedding. The gpt-4o-mini model is used to generate a response to a prompt. The vector embedding model is used to compare the similarity of two prompts.

Extending the String Class

```
1  import Foundation
2  import NaturalLanguage
3
4  // String utilities:
5
6  extension String {
7      func removeCharacters(from forbiddenChars: CharacterSet) -> String {
8          let passed = self.unicodeScalars.filter { !forbiddenChars.contains($0) }
9          return String(String.UnicodeScalarView(passed))
10     }
11
12     func removeCharacters(from: String) -> String {
13         return removeCharacters(from: CharacterSet(charactersIn: from))
```

[49] https://github.com/mark-watson/Docs_QA_Swift

```
    }
    func plainText() -> String {
        return self.removeCharacters(from:
                                     "\"`()%$#@[]{}<>").replacingOccurrences(of: "\n\
",
                                     with: " ")
    }
}
```

Implementing a Local Vector Database for Document Embeddings

```
let openai_key = ProcessInfo.processInfo.environment["OPENAI_KEY"]!

let openAiHost = "https://api.openai.com/v1/embeddings"

func openAiHelper(body: String)  -> String {
    var ret = ""
    var content = "{}"
    let requestUrl = URL(string: openAiHost)!
    var request = URLRequest(url: requestUrl)
    request.httpMethod = "POST"
    request.httpBody = body.data(using: String.Encoding.utf8);
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.setValue("Bearer " + openai_key, forHTTPHeaderField: "Authorization")
    let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
        if let error = error {
            print("-->> Error accessing OpenAI servers: \(error)")
            return
        }
        if let data = data, let s = String(data: data, encoding: .utf8) {
            content = s
            //print("** s=", s)
            CFRunLoopStop(CFRunLoopGetMain())
        }
    }
    task.resume()
    CFRunLoopRun()
    let c = String(content)
    let i1 = c.range(of: "\"embedding\":")
    if let r1 = i1 {
```

```
        let i2 = c.range(of: "]")
        if let r2 = i2 {
            ret = String(String(String(c[r1.lowerBound..<r2.lowerBound]).dropFirst(1\
5)).dropLast(2))
        }
    }
    return ret
}

public func embeddings(someText: String) -> [Float] {
    let body: String = "{\"input\": \"" + someText + "\", \"model\": \"text-embeddin\
g-ada-002\" }"
    return readList(openAiHelper(body: body))
}

func dotProduct(_ list1: [Float], _ list2: [Float]) -> Float {
    if list1.count != list2.count {
        //fatalError("Lists must have the same length.")
        print("WARNING: Lists must have the same length: \(list1.count) != \(list2.c\
ount)")
        return 0.0
    }

    var result: Float = 0

    for i in 0..<list1.count {
        result += list1[i] * list2[i]
    }

    return result
}
```

The source file contains example code for creating embeddings and using dot product work to find semantic similarity:

```
let emb1 = embeddings(someText: "John bought a new car")
let emb2 = embeddings(someText: "Sally drove to the store")
let emb3 = embeddings(someText: "The dog saw a cat")
let dotProductResult1 = dotProduct(emb1, emb2)
print(dotProductResult1)
let dotProductResult2 = dotProduct(emb1, emb3)
print(dotProductResult2)
```

The output is:

```
0.8416926
0.79411536
```

For this example, we use an in-memory store of embedding vectors and chunk text. A text document is broken into smaller chunks of text. Each chunk is embedded and stored in the embeddingsStore. The chunk text is stored in the chunks array. The embeddingsStore and chunks array are used to find the most similar chunk to a prompt. The most similar chunk is used to generate a response to the prompt.

```
var embeddingsStore: Array<[Float]> = Array()
var chunks: Array<String> = Array()

func addEmbedding(_ embedding: [Float]) {
    embeddingsStore.append(embedding)
    //print("Added embedding: count=\(embeddingsStore.count)  \(embedding)")
}

func addChunk(_ chunk: String) {
    chunks.append(chunk)
}
```

Create Local Embeddings Vectors From Local Text Files With OpenAI GPT APIs

```
func readList(_ input: String) -> [Float] {
    return input.split(separator: ",\n").compactMap {
        Float($0.trimmingCharacters(in: .whitespaces))
    }
}

let fileManager = FileManager.default
let currentDirectoryURL = URL(fileURLWithPath: fileManager.currentDirectoryPath)
let dataDirectoryURL = currentDirectoryURL.appendingPathComponent("data")

// Top level code expression to process all *.txt files in the data/ directory:

do {
    let directoryContents = try fileManager.contentsOfDirectory(at: dataDirectoryURL\
, includingPropertiesForKeys: nil)
    let txtFiles = directoryContents.filter { $0.pathExtension == "txt" }
    for txtFile in txtFiles {
        let content = try String(contentsOf: txtFile)
        let chnks = segmentTextIntoChunks(text: content.plainText(),
                                          max_chunk_size: 100)
        for chunk in chnks {
            let embedding = embeddings(someText: chunk)
            if embedding.count > 0 {
                addEmbedding(embedding)
                addChunk(chunk)
            }
        }
    }
} catch {
}

func segmentTextIntoSentences(text: String) -> [String] {
    let tokenizer = NLTokenizer(unit: .sentence)
    tokenizer.string = text
    let sentences = tokenizer.tokens(for: text.startIndex..<text.endIndex).map {
       token -> String in
         return String(text[token.lowerBound..<token.upperBound])
    }
    return sentences
}

func segmentTextIntoChunks(text: String, max_chunk_size: Int) -> [String] {
    let sentences = segmentTextIntoSentences(text: text)
```

```
    var chunks: Array<String> = Array()
    var currentChunk = ""
    var currentChunkSize = 0
    for sentence in sentences {
        if currentChunkSize + sentence.count < max_chunk_size {
            currentChunk += sentence
            currentChunkSize += sentence.count
        } else {
            chunks.append(currentChunk)
            currentChunk = sentence
            currentChunkSize = sentence.count
        }
    }
    return chunks
}
```

Using Local Embeddings Vector Database With OpenAI GPT APIs

We use the OpenAI QA API using gpt-4o-mini model (reformatted to fit the page width):

```
let openAiQaHost = "https://api.openai.com/v1/chat/completions"

func openAiQaHelper(body: String)  -> String {
    var ret = ""
    var content = "{}"
    let requestUrl = URL(string: openAiQaHost)!
    var request = URLRequest(url: requestUrl)
    request.httpMethod = "POST"
    request.httpBody = body.data(using: String.Encoding.utf8);
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.setValue("Bearer " + openai_key, forHTTPHeaderField: "Authorization")
    let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
        if let error = error {
            print("-->> Error accessing OpenAI servers: \(error)")
            return
        }
        if let data = data, let s = String(data: data, encoding: .utf8) {
            content = s
            CFRunLoopStop(CFRunLoopGetMain())
```

```
        }
    }
    task.resume()
    CFRunLoopRun()
    let c = String(content)
    //print("DEBUG response c:", c)
    // pull returned content for string instead of using a
    // JSON parser:
    let i1 = c.range(of: "\"content\":")
    if let r1 = i1 {
        let i2 = c.range(of: "\"}")
        if let r2 = i2 {
            ret = String(
                    String(
                      String(c[r1.lowerBound..<r2.lowerBound])
                        .dropFirst(11)))
        }
    }
    return ret
}

func questionAnswering(context: String, question: String) -> String {
    let body = "{ \"model\": \"gpt-3.5-turbo\", \"messages\": [ {\"role\": \"system\\
", \"content\": \"" +
      context + "\"}, {\"role\": \"user\", \"content\": \"" + question + "\"}]}"

    //print("DEBUG body:", body)

    let answer = openAiQaHelper(body: body)
    if let i1 = answer.range(of: "\"content\":") {
        // variable answer is a string containing JSON. We want to extract the value\
 of the "content" key and we do so without using a JSON parser.
        return String(answer[answer.startIndex..<i1.lowerBound])
    }
    return answer
}

//  Top level query interface:

func query(_ query: String) -> String {
    let queryEmbedding = embeddings(someText: query)
    var contextText = ""
    for i in 0..<embeddingsStore.count {
```

```
        let dotProductResult = dotProduct(queryEmbedding, embeddingsStore[i])
        if dotProductResult > 0.8 {
            contextText.append(chunks[i])
            contextText.append(" ")
        }
    }
    //print("\n\n+++++++ contextText = \(contextText)\n\n")
    let answer = questionAnswering(context: contextText, question: query)
    //print("* * debug: query: ", query)
    //print("* * debug: answer:", answer)
    return answer

}

print(query("What is the history of chemistry?"))
print(query("What is the definition of sports?"))
print(query("What is the microeconomics?"))
```

The output for these three questions looks like:

```
The history of chemistry dates back to ancient times when people began to manipulate\
 materials to produce useful products. The ancient Egyptians were skilled in metallu\
rgy and used various chemicals to embalm bodies. The Greeks were interested in theor\
ies of matter and sought to understand the nature of substances.\n\nDuring the Middl\
e Ages, alchemy became popular, with alchemists seeking to transform base metals int\
o gold and searching for an elixir of life. While alchemy was considered a pseudosci\
ence, it did lead to important discoveries such as the distillation of alcohol and t\
he discovery of various acids.\n\nThe Scientific Revolution of the 17th century brou\
ght about significant changes in chemistry. The work of Robert Boyle, Antoine Lavois\
ier, and others laid the foundation for modern chemistry. Lavoisier is considered th\
e father of modern chemistry for his work in establishing the law of conservation of\
 mass, which states that matter cannot be created or destroyed.\n\nThe 19th century \
saw the development of organic chemistry, as scientists sought to understand the che\
mistry of carbon-based compounds, which make up many biological molecules. The 20th \
century brought about significant advances in analytical chemistry, as well as the d\
evelopment of quantum mechanics and the discovery of the structure of DNA, which rev\
olutionized the field of biochemistry.\n\nToday, chemistry plays a critical role in \
fields such as medicine, agriculture, materials science, and environmental science.


Sports can be defined as activities involving physical athleticism, physical dexteri\
ty, and governed by rules to ensure fair competition and consistent adjudication of \
the winner. The term \"sport\" originally meant leisure, but it now primarily refers\
```

```
 to physical activities that involve competition at various levels of skill and prof\
iciency. Some organizations also include all physical activity and exercise in the d\
efinition of sport.


Microeconomics is a branch of economics that focuses on the behavior and decision-ma\
king of individual units within an economy, such as households, firms, and industrie\
s. It examines how these units interact in various markets to determine the prices o\
f goods and services and how resources are allocated efficiently. Microeconomics als\
o considers the role of government policies and regulations in influencing these int\
eractions and outcomes. Topics studied in microeconomics include supply and demand, \
market structures, consumer behavior, production and cost analysis, and welfare anal\
ysis.
```

Wrap Up for Using Local Embeddings Vector Database to Enhance the Use of GPT3 APIs With Local Documents

As I write this in early April 2023, I have been working almost exclusively with OpenAI APIs for the last year and using the Python libraries for LangChain and LlamaIndex for the last three months.

I started writing the examples in this chapter for my own use, implementing a tiny subset of the LangChain and LlamaIndex libraries in Swift in order to write efficient command line utilities for creating local embedding vector data stores and for interactive chat using my own data.

By writing about my “scratching my own itch” command line experiments here I hope that I get pull requests for https://github.com/mark-watson/Docs_QA_Swift from readers who are interested in helping to extend this code with new functionality.

Part 4: Knowledge Representation and Data Acquisition

In this part we cover:

- Introduction to the semantic web and linked data
- A general discussion of Knowledge Representation
- Create Knowledge Graphs from text input
- Knowledge Graph Explorer application

Linked Data and the Semantic Web

Tim Berners Lee, James Hendler, and Ora Lassila wrote in 2001 an article for Scientific American where they introduced the term Semantic Web. Here I do not capitalize semantic web and use the similar term linked data somewhat interchangeably with semantic web.

In the same way that the web allows links between related web pages, linked data supports linking associated data on the web together. I view linked data as a relatively simple way to specify relationships between data sources on the web while the semantic web has a much larger vision: the semantic web has the potential to be the entirety of human knowledge represented as data on the web in a form that software agents can work with to answer questions, perform research, and to infer new data from existing data.

While the "web" describes information for human readers, the semantic web is meant to provide structured data for ingestion by software agents. This distinction will be clear as we compare WikiPedia, made for human readers, with DBPedia which uses the info boxes on WikiPedia topics to automatically extract RDF data describing WikiPedia topics. Let's look at the WikiPedia topic for the town I live in Sedona, Arizona, and show how the info box on the English version of the WikiPedia topic page for Sedona https://en.wikipedia.org/wiki/Sedona,_Arizona[50] maps to the DBPedia page http://dbpedia.org/page/Sedona,_Arizona[51]. Please open both of these WikiPedia and DBPedia URIs in two browser tabs and keep them open for reference.

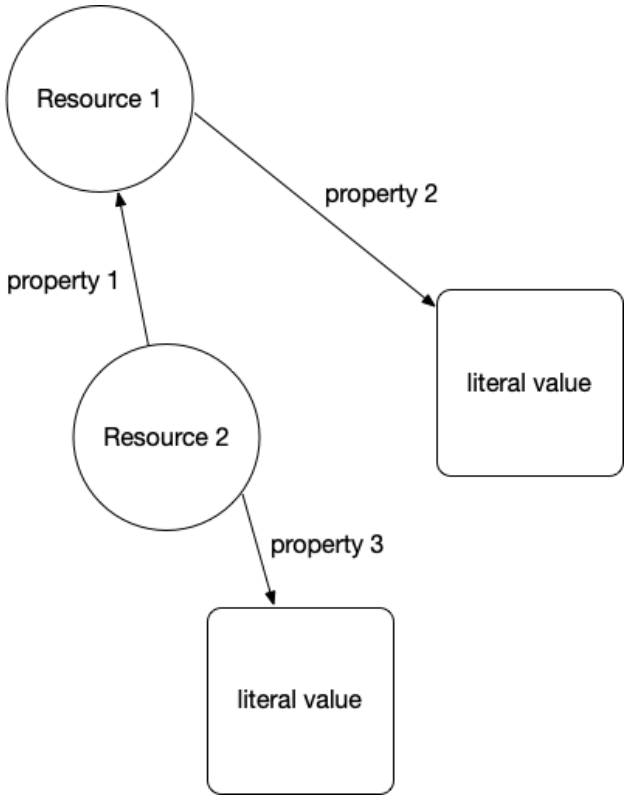
I assume that the format of the WikiPedia page is familiar so let's look at the DBPedia page for Sedona that in human readble form shows the RDF statements with Sedona Arizona as the subject. RDF is used to model and represent data. RDF is defined by three values so an instance of an RDF statement is called a *triple* with three parts:

- subject: a URI (also referred to as a "Resource")
- property: a URI (also referred to as a "Resource")
- value: a URI (also referred to as a "Resource") or a literal value (like a string or a number with optional units)

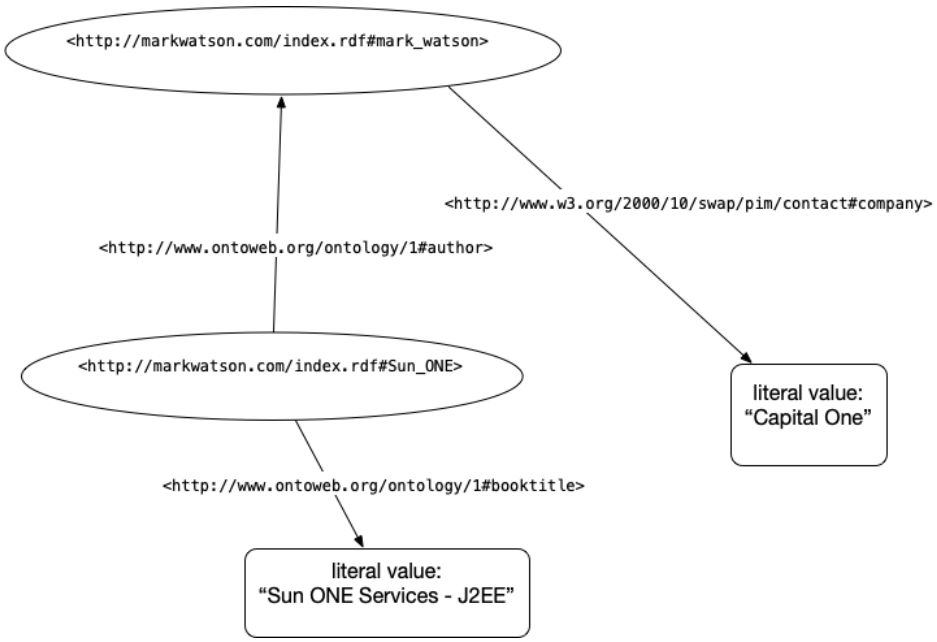
The subject for each Sedona related triple is the above URI for the DBPedia human readable page. The subject and property references in an RDF triple will almost always be a URI that can ground an entity to information on the web. The human readable page for Sedona lists several properties and the values of these properties. One of the properties is "dbo:areaCode" where "dbo" is a name space reference (in this case for a DatatypeProperty[52]).

[50] https://en.wikipedia.org/wiki/Sedona,_Arizona
[51] http://dbpedia.org/page/Sedona,_Arizona
[52] http://www.w3.org/2002/07/owl#DatatypeProperty

The following two figures show an abstract representation of linked data and then a sample of linked data with actual web URIs for resources and properties:



Abstract RDF representation with 2 Resources, 2 literal values, and 3 Properties



Concrete example using RDF seen in last chapter showing the RDF representation with 2 Resources, 2 literal values, and 3 Properties

We will use the SPARQL query language (SPARQL for RDF data is similar to SQL for relational database queries). Let's look at an example using the RDF in the last figure:

```
"select ?v where { <http://markwatson.com/index.rdf#Sun_ONE>
                   <http://www.ontoweb.org/ontology/1#booktitle>
                   ?v }
```

This query should return the result "Sun ONE Services - J2EE". If you wanted to query for all URI resources that are books with the literal value of their titles, then you can use:

```
"select ?s ?v where { ?s
                      <http://www.ontoweb.org/ontology/1#booktitle>
                      ?v }
```

Note that **?s** and **?v** are arbitrary query variable names, here standing for "subject" and "value". You can use more descriptive variable names like:

```
"select ?bookURI ?bookTitle where
    { ?bookURI
      <http://www.ontoweb.org/ontology/1#booktitle>
      ?bookTitle }
```

We will be diving a little deeper into RDF examples in the next chapter when we write a tool for using RDF data from DBPedia to find information about entities (e.g., people, places, organizations)

and the relationships between entities. For now I want you to understand the idea of RDF statements represented as triples, that web URIs represent things, properties, and sometimes values, and that URIs can be followed manually (often called "dereferencing") to see what they reference in human readable form.

Understanding the Resource Description Framework (RDF)

Text data on the web has some structure in the form of HTML elements like headers, page titles, anchor links, etc. but this structure is too imprecise for general use by software agents. RDF is a method for encoding structured data in a more precise way.

RDF specifies graph structures and can be serialized for storage or for service calls in XML, Turtle, N3, and other formats. I like the Turtle format and suggest that you pause reading this book for a few minutes and look at this World Wide Web Consortium Turtle RDF primer at https://www.w3.org/2007/02/turtle/primer/[53].

Frequently Used Resource Namespaces

The following standard namespaces are frequently used:

- RDF https://www.w3.org/TR/rdf-syntax-grammar/[54]
- RDFS https://www.w3.org/TR/rdf-schema/[55]
- OWL http://www.w3.org/2002/07/owl#[56]
- XSD http://www.w3.org/2001/XMLSchema#[57]
- FOAF http://xmlns.com/foaf/0.1/[58]
- SKOS http://www.w3.org/2004/02/skos/core#[59]
- DOAP http://usefulinc.com/ns/doap#[60]
- DC http://purl.org/dc/elements/1.1/[61]
- DCTERMS http://purl.org/dc/terms/[62]
- VOID http://rdfs.org/ns/void#[63]

Let's look into the Friend of a Friend (FOAF) namespace. Click on the above link for FOAF http://xmlns.com/foaf/0.1/[64] and find the definitions for the FOAF Core:

[53] https://www.w3.org/2007/02/turtle/primer/
[54] https://www.w3.org/TR/rdf-syntax-grammar/
[55] https://www.w3.org/TR/rdf-schema/
[56] http://www.w3.org/2002/07/owl#
[57] http://www.w3.org/2001/XMLSchema#
[58] http://xmlns.com/foaf/0.1/
[59] http://www.w3.org/2004/02/skos/core#
[60] http://usefulinc.com/ns/doap#
[61] http://purl.org/dc/elements/1.1/
[62] http://purl.org/dc/terms/
[63] http://rdfs.org/ns/void#
[64] http://xmlns.com/foaf/0.1/

```
    Agent
    Person
    name
    title
    img
    depiction (depicts)
    familyName
    givenName
    knows
    based_near
    age
    made (maker)
    primaryTopic (primaryTopicOf)
    Project
    Organization
    Group
    member
    Document
    Image
```

and for the Social Web:

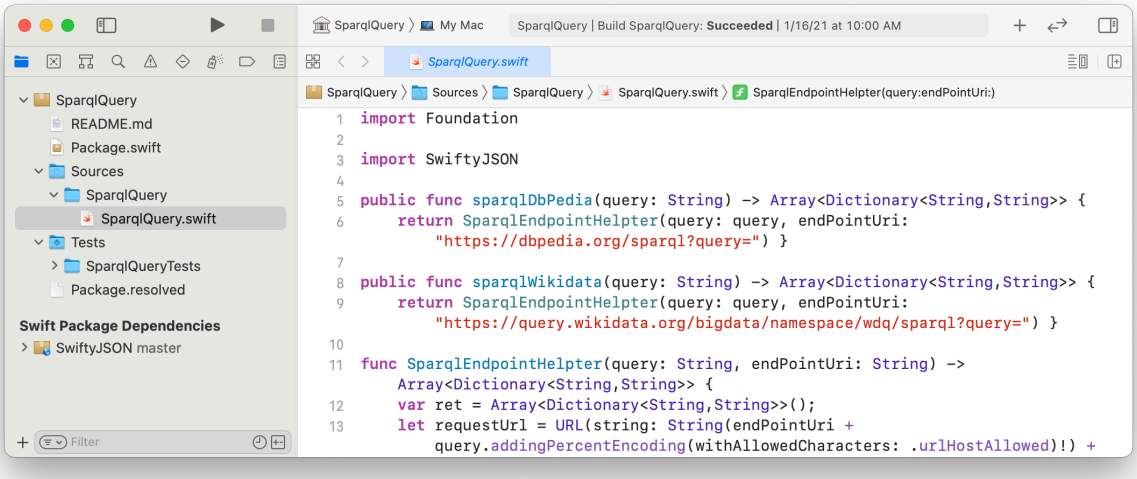
```
mbox
homepage
weblog
openid
jabberID
mbox_sha1sum
interest
topic_interest
topic (page)
workplaceHomepage
workInfoHomepage
schoolHomepage
publications
currentProject
pastProject
account
OnlineAccount
accountName
accountServiceHomepage
PersonalProfileDocument
tipjar
```

```
sha1
thumbnail
logo
```

You now have seen a few common Schemas for RDF data. Another Schema that is widely used for annotating web sites that we won't need for our examples here, is schema.org[65].

Understanding the SPARQL Query Language

For the purposes of the material in this book, the two sample SPARQL queries here are sufficient for you to get started using my SPARQL library https://github.com/mark-watson/SparqlQuery_swift[66] with arbitrary RDF data sources and simple queries.



My Swift SPARQL library open in Xcode

The Apache Foundation has a good introduction to SPARQL[67] that I refer you to for more information.

Semantic Web and Linked Data Wrap Up

In the next chapter we will use natural language processing to extract structured information from raw text from SPARQL queries. We will be using my Swift SPARQL library https://github.com/mark-watson/SparqlQuery_swift[68] as well as two pre-trained CoreML deep learning models.

[65] https://schema.org
[66] https://github.com/mark-watson/SparqlQuery_swift
[67] https://jena.apache.org/tutorials/sparql.html
[68] https://github.com/mark-watson/SparqlQuery_swift

Example Application: iOS and macOS Versions of my KnowledgeBookNavigator

I used many of the techniques discussed in this book, the Swift language, and the SwiftUI user interface framework to develop Swift version of my Knowledge Graph Navigator application for macOS. I originally wrote this as an example program in Common Lisp for another book project.

The GitHub repository for the KGN example is https://github.com/mark-watson/KGN[69]. I copied the code from my stand-alone Swift libraries to this example to make it self contained. The easiest way to browse the source code is to open this project in Xcode.

I submitted the KGN app that we discuss in this chapter to Apple's store and is available as a macOS app. If you load this project into Xcode, you can also build and run the iOS and iPadOS targets.

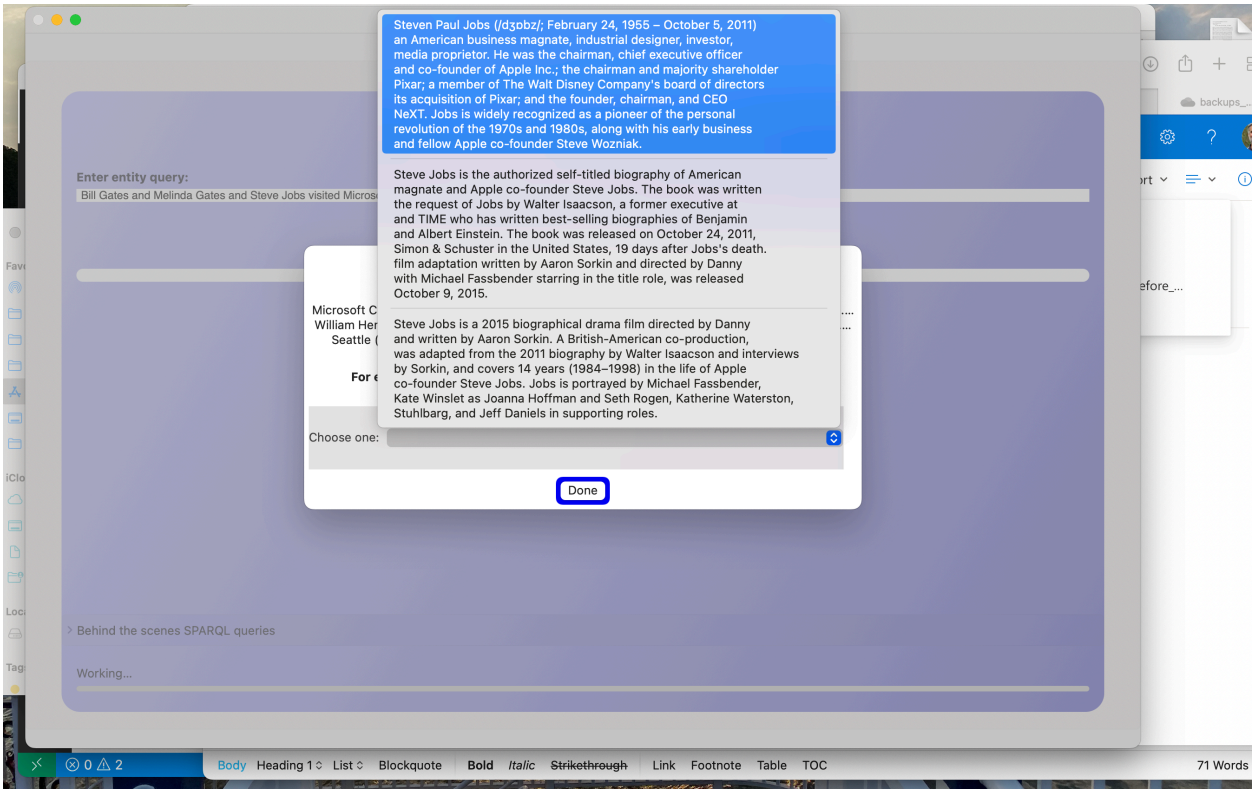
You will need to have read through the last chapter on semantic web and linked data technologies to understand this example because quite a lot of the code has embedded SPARQL queries to get information from DBPedia.org[70].

The other major part of this app is a slightly modified version of Apple's question answering (QA) example using the BERT model in CoreML. Apple's code is in the subdirectory **AppleBERT**. Please read the README file for this project and follow the directions for downloading and using Apple's model and vocabulary file.

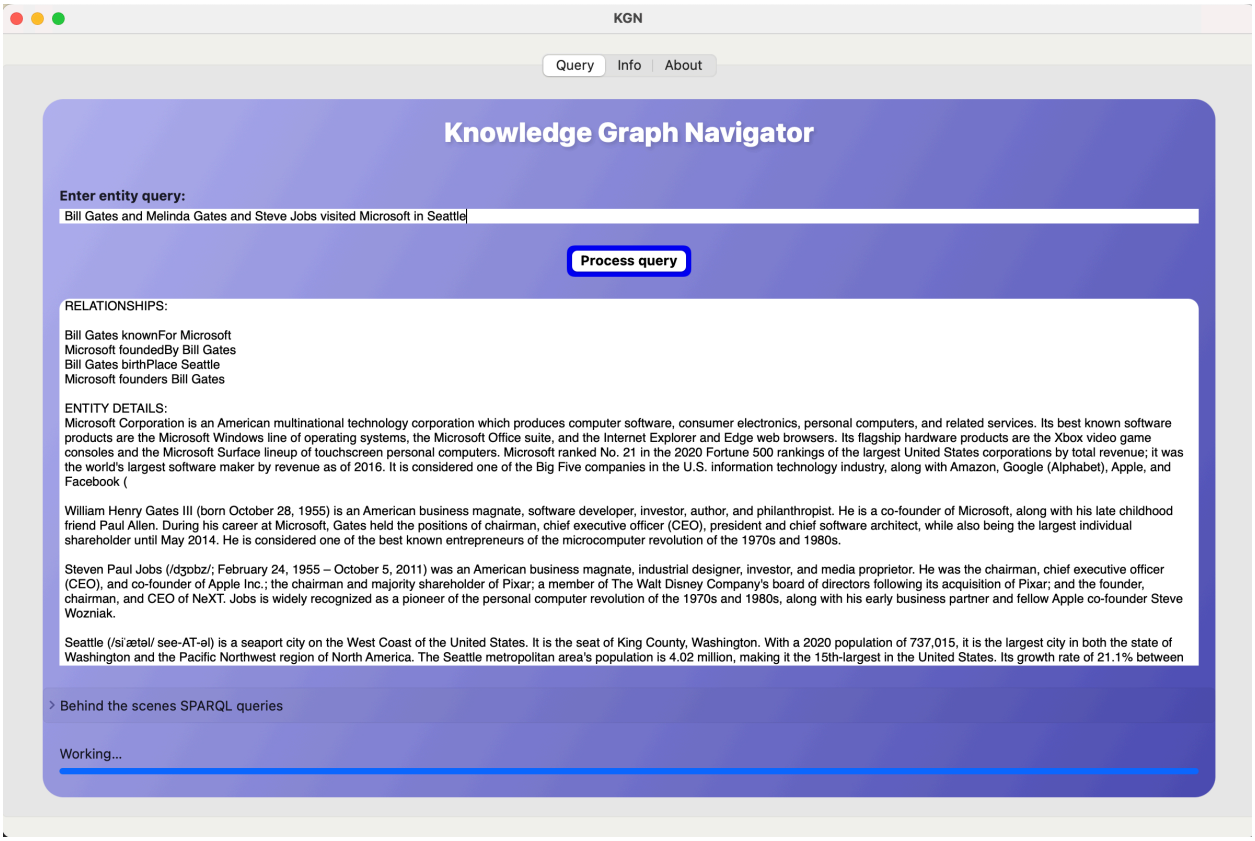
Screen Shots of macOS Application

In the first screenshot seen below, I had entered query text that included "Steve Jobs" and the popup list selector is used to let the user select which "Steve Jobs" entity from DBPedia that they want to use.

[69] https://github.com/mark-watson/KGN
[70] https://dbpedia.org



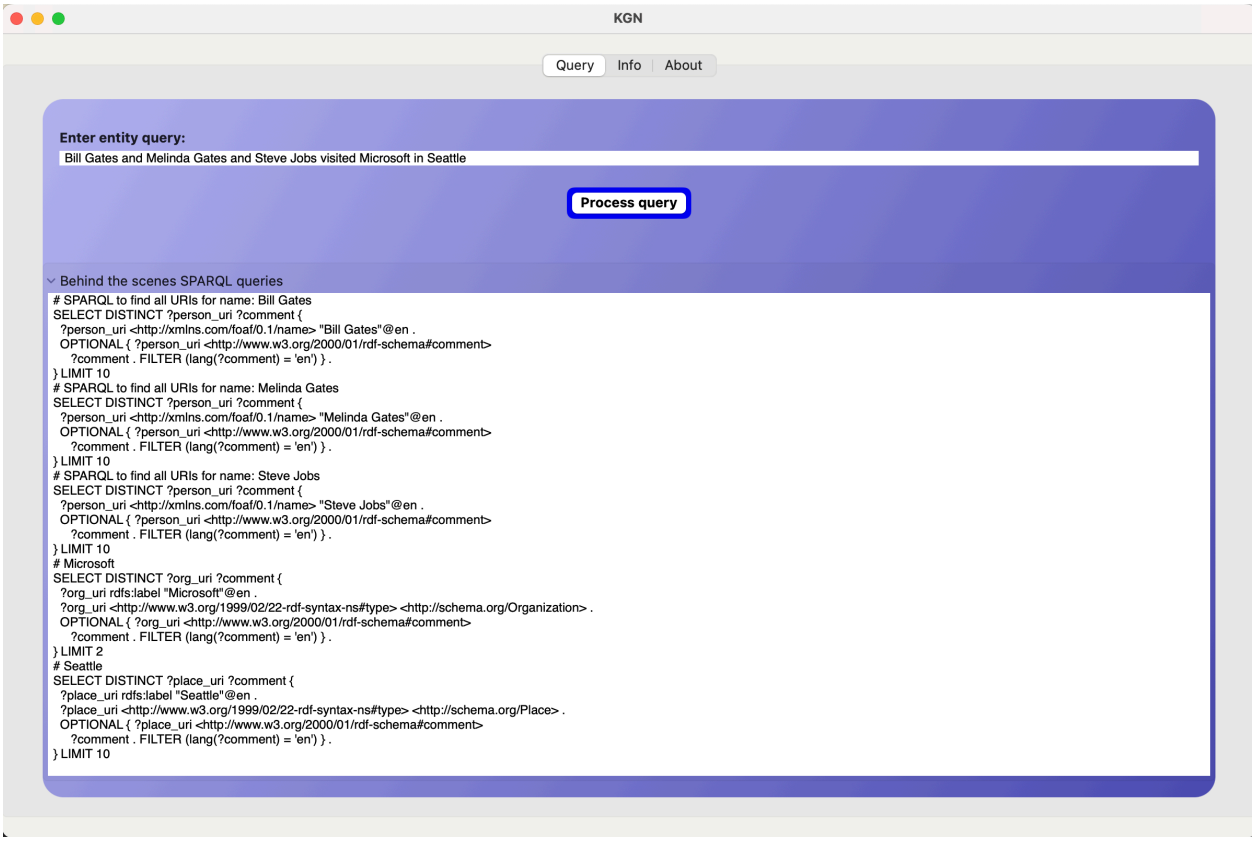
Entered query and KGN is asking user to disambiguate which “Steve Jobs” they want information for



Showing results

The previous screenshot shows the results to the query displayed as English text.

Notice the app prompt “Behind the scenes SPARQL queries” near the bottom of the app window. If you click on this field then the SPARQL queries used to answer the question are shown, as on the next screenshot:



Showing SPARQL queries used to gather data

Application Code Listings

I will list some of the code for this example application and I suggest that you, dear reader, also open this project in Xcode in order to navigate the sample code and more carefully read through it.

SPARQL

I introduced you to the use of SPARQL in the last chapter. This library can be used by adding a reference to the **Project.swift** file for this project. You can also clone the GitHub repository https://github.com/mark-watson/Nlp_swift[71] to have the source code for local viewing and modification and I have copied the code into the KGN project.

The file **SparqlQuery.swift** is shown here:

[71] https://github.com/mark-watson/Nlp_swift

```
import Foundation

public func sparqlDbPedia(query: String) -> Array<Dictionary<String,String>> {
    return SparqlEndpointHelpter(query: query,
        endPointUri: "https://dbpedia.org/sparql?query=") }

public func sparqlWikidata(query: String) -> Array<Dictionary<String,String>> {
    return SparqlEndpointHelpter(query: query,
        endPointUri:
          "https://query.wikidata.org/bigdata/namespace/wdq/sparql?query=") }

public func SparqlEndpointHelpter(query: String,
                                  endPointUri: String) ->
                            Array<Dictionary<String,String>> {
    var ret = Set<Dictionary<String,String>>();
    var content = "{}"

    let maybeString = cacheLookupQuery7(key: query)
    if maybeString?.count ?? 0 > 0 {
        content = maybeString ?? ""
    } else {
        let requestUrl = URL(string: String(endPointUri + query.addingPercentEncodin\
g(withAllowedCharacters:
          .urlHostAllowed)!) + "&format=json")!
        do { content = try String(contentsOf: requestUrl) }
          catch let error { print(error) }
    }
    let json = try? JSONSerialization.jsonObject(with: Data(content.utf8),
                                                 options: [])
    if let json2 = json as! Optional<Dictionary<String, Any?>> {
        if let head = json2["head"] as? Dictionary<String, Any> {
            if let xvars = head["vars"] as! NSArray? {
                if let results = json2["results"] as? Dictionary<String, Any> {
                    if let bindings = results["bindings"] as! NSArray? {
                        if bindings.count > 0 {
                            for i in 0...(bindings.count-1) {
                                if let first_binding =
                                bindings[i] as? Dictionary<String,
                                Dictionary<String,String>> {
                                    var ret2 = Dictionary<String,String>();
                                    for key in xvars {
                                        let key2 : String = key as! String
                                        if let vals = (first_binding[key2]) {
```

```
                                        let vv : String = vals["value"] ?? "err2"
                                        ret2[key2] = vv } }
                                if ret2.count > 0 {
                                    ret.insert(ret2)
                                }}}}}}}}}
    return Array(ret) }
```

The file **QueryCache.swift** contains code written by Khoa Pham (MIT License) that can be found in the GitHub repository https://github.com/onmyway133/EasyStash[72]. This file is used to cache SPARQL queries and the results. In testing this application I noticed that there were many repeated queries to DBPedia so I decided to cache results. Here is the simple API I added on top of Khoa Pham's code:

```
//  Created by khoa on 27/05/2019.
//  Copyright © 2019 Khoa Pham. All rights reserved. MIT License.
//  https://github.com/onmyway133/EasyStash
//

import Foundation

//      Mark's simple wrapper:

var storage: Storage? = nil

public func cacheStoreQuery(key: String, value: String) {
    do { try storage?.save(object: value, forKey: key) } catch {}
}
public func cacheLookupQuery7(key: String) -> String? {
    // optional DEBUG code: clear cache
    //do { try storage?.removeAll() } catch { print("ERROR CLEARING CACHE") }
    do {
        return try storage?.load(forKey: key, as: String.self)
    } catch { return "" }
}

// remaining code not shown for brevity.
```

The code in file **GenerateSparql.swift** is used to generate queries for DBPedia. The line-wrapping for embedded SPARQL queries in the next code section is difficult to read so you may want to open the source file in Xcode. Please note that the KGN application prints out the SPARQL queries used to fetch information from DBPedia. The embedded SPARQL query templates used here have variable slots that filled in at runtime to customize the queries.

[72] https://github.com/onmyway133/EasyStash

```
//
//  GenerateSparql.swift
//  KGNbeta1
//
//  Created by Mark Watson on 2/28/20.
//  Copyright © 2021 Mark Watson. All rights reserved.
//

import Foundation

public func uri_to_display_text(uri: String)
                                    -> String {
    return uri.replacingOccurrences(of: "http://dbpedia.org/resource/Category/",
        with: "").
      replacingOccurrences(of: "http://dbpedia.org/resource/",
        with: "").
         replacingOccurrences(of: "_", with: " ")
}

public func get_SPARQL_for_finding_URIs_for_PERSON_NAME(nameString: String)
                                            -> String {
    return
        "# SPARQL to find all URIs for name: " +
        nameString + "\nSELECT DISTINCT ?person_uri ?comment {\n" +
        "  ?person_uri <http://xmlns.com/foaf/0.1/name> \"" +
        nameString + "\"@en .\n" +
        "  OPTIONAL { ?person_uri <http://www.w3.org/2000/01/rdf-schema#comment>\n" +
        "     ?comment . FILTER (lang(?comment) = 'en') } .\n" +
        "} LIMIT 10\n"
}

public func get_SPARQL_for_PERSON_URI(aURI: String) -> String {
    return
        "# <" + aURI + ">\nSELECT DISTINCT ?comment (GROUP_CONCAT(DISTINCT ?birthpla\
ce; SEPARATOR=' | ') AS ?birthplace)\n  (GROUP_CONCAT(DISTINCT ?almamater; SEPARATOR\
=' | ') AS ?almamater) (GROUP_CONCAT(DISTINCT ?spouse; SEPARATOR=' | ') AS ?spouse) \
{\n" +
        "  <" + aURI + "> <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment .\
 FILTER  (lang(?comment) = 'en') .\n" +
        "  OPTIONAL { <" + aURI + "> <http://dbpedia.org/ontology/birthPlace> ?birth\
place } .\n" +
        "  OPTIONAL { <" + aURI + "> <http://dbpedia.org/ontology/almaMater> ?almama\
ter } .\n" +
```

```
        "  OPTIONAL { <" + aURI + "> <http://dbpedia.org/ontology/spouse> ?spouse } \
.\n" +
        "} LIMIT 5\n"
}

public func get_display_text_for_PERSON_URI(personURI: String) -> [String] {
    var ret: String = "\(uri_to_display_text(uri: personURI))\n\n"
    let person_details_sparql = get_SPARQL_for_PERSON_URI(aURI: personURI)
    let person_details = sparqlDbPedia(query: person_details_sparql)

    for pd in person_details {
        //let comment = pd["comment"]
        ret.append("\(pd["comment"] ?? "")\n\n")
        let subject_uris = pd["subject_uris"]
        let uri_list: [String] = subject_uris?.components(separatedBy: " | ") ?? []
        //ret.append("<ul>\n")
        for u in uri_list {
            let subject = uri_to_display_text(uri: u)
            ret.append("\(subject)\n") }
        //ret.append("</ul>\n")
        if let spouse = pd["spouse"] {
            if spouse.count > 0 {
                ret.append("Spouse: \(uri_to_display_text(uri: spouse))\n") } }
        if let almamater = pd["almamater"] {
            if almamater.count > 0 {
                ret.append("Almamater: \(uri_to_display_text(uri: almamater))\n") } }
        if let birthplace = pd["birthplace"] {
            if birthplace.count > 0 {
                ret.append("Birthplace: \(uri_to_display_text(uri: birthplace))\n") \
} }
    }
    return ["# SPARQL for a specific person:\n" + person_details_sparql, ret]
}

//     "  ?place_uri <http://xmlns.com/foaf/0.1/name> \"" + placeString + "\"@en .\n\
" +

public func get_SPARQL_for_finding_URIs_for_PLACE_NAME(placeString: String)
                                                -> String {
    return
        "# " + placeString + "\nSELECT DISTINCT ?place_uri ?comment {\n" +
        "  ?place_uri rdfs:label \"" + placeString + "\"@en .\n" +
        "  ?place_uri <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://sche\
```

```
ma.org/Place> .\n" +
        "  OPTIONAL { ?place_uri <http://www.w3.org/2000/01/rdf-schema#comment>\n" +
        "     ?comment . FILTER (lang(?comment) = 'en') } .\n" +
        "} LIMIT 10\n"
}

public func get_SPARQL_for_PLACE_URI(aURI: String) -> String {
    return
        "# <" + aURI + ">\nSELECT DISTINCT ?comment (GROUP_CONCAT(DISTINCT ?subject_\
uris; SEPARATOR=' | ') AS ?subject_uris) {\n" +
        "  <" + aURI + "> <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment .\
 FILTER  (lang(?comment) = 'en') .\n" +
        "  OPTIONAL { <" + aURI + "> <http://purl.org/dc/terms/subject> ?subject_uri\
s } .\n" +
        "} LIMIT 5\n"
}

public func get_HTML_for_place_URI(placeURI: String) -> String {
    var ret: String = "<h2>" + placeURI + "</h2>\n"
    let place_details_sparql = get_SPARQL_for_PLACE_URI(aURI: placeURI)
    let place_details = sparqlDbPedia(query: place_details_sparql)

    for pd in place_details {
        //let comment = pd["comment"]
        ret.append("<p><strong>\(pd["comment"] ?? "")</strong></p>\n")
        let subject_uris = pd["subject_uris"]
        let uri_list: [String] = subject_uris?.components(separatedBy: " | ") ?? []
        ret.append("<ul>\n")
        for u in uri_list {
            let subject = u.replacingOccurrences(of: "http://dbpedia.org/resource/Ca\
tegory:", with: "").replacingOccurrences(of: "_", with: " ").replacingOccurrences(of\
: "-", with: " ")
            ret.append("  <li>\(subject)</li>\n")
        }
        ret.append("</ul>\n")
    }
    return ret
}

public func get_SPARQL_for_finding_URIs_for_ORGANIZATION_NAME(orgString: String) -> \
String {
    return
        "# " + orgString + "\nSELECT DISTINCT ?org_uri ?comment {\n" +
```

```
        "  ?org_uri rdfs:label \"" + orgString + "\"@en .\n" +
        "  ?org_uri <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema\
.org/Organization> .\n" +
        "  OPTIONAL { ?org_uri <http://www.w3.org/2000/01/rdf-schema#comment>\n" +
        "     ?comment . FILTER (lang(?comment) = 'en') } .\n" +
        "} LIMIT 2\n"
}
```

The file **AppSparql** contains more utility functions for getting entity and relationship data from DBPedia:

```
//  AppSparql.swift
//  Created by ML Watson on 7/18/21.

import Foundation

let detailSparql = """
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
select ?entity ?label ?description ?comment where {
    ?entity rdfs:label "<name>"@en .
    ?entity schema:description ?description . filter (lang(?description) = 'en') . f\
ilter(!regex(?description,"Wikimedia disambiguation page")) .
 } limit 5000
"""

let personSparql = """
  select ?uri ?comment {
      ?uri <http://xmlns.com/foaf/0.1/name> "<name>"@en .
      ?uri <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment .
          FILTER  (lang(?comment) = 'en') .
  }
"""


let personDetailSparql = """
SELECT DISTINCT ?label ?comment

     (GROUP_CONCAT (DISTINCT ?birthplace; SEPARATOR=' | ') AS ?birthplace)
     (GROUP_CONCAT (DISTINCT ?almamater; SEPARATOR=' | ') AS ?almamater)
     (GROUP_CONCAT (DISTINCT ?spouse; SEPARATOR=' | ') AS ?spouse) {
       <name> <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment .
       FILTER  (lang(?comment) = 'en') .
     OPTIONAL { <name> <http://dbpedia.org/ontology/birthPlace> ?birthplace } .
```

```
      OPTIONAL { <name> <http://dbpedia.org/ontology/almaMater> ?almamater } .
      OPTIONAL { <name> <http://dbpedia.org/ontology/spouse> ?spouse } .
      OPTIONAL { <name>  <http://www.w3.org/2000/01/rdf-schema#label> ?label .
         FILTER  (lang(?label) = 'en') }
} LIMIT 10
"""

let placeSparql = """
SELECT DISTINCT ?uri ?comment WHERE {
   ?uri rdfs:label "<name>"@en .
   ?uri <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment .
   FILTER (lang(?comment) = 'en') .
   ?place <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/Place\
> .
} LIMIT 80
"""

let organizationSparql = """
SELECT DISTINCT ?uri ?comment WHERE {
   ?uri rdfs:label "<name>"@en .
   ?uri <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment .
   FILTER (lang(?comment) = 'en') .
   ?uri <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/Organiz\
ation> .
} LIMIT 80
"""

func entityDetail(name: String) -> [Dictionary<String,String>] {
    var ret: [Dictionary<String,String>] = []
    let sparql = detailSparql.replacingOccurrences(of: "<name>", with: name)
    print(sparql)
    let r = sparqlDbPedia(query: sparql)
    r.forEach { result in
        print(result)
        ret.append(result)
    }
    return ret
}

func personDetail(name: String) -> [Dictionary<String,String>] {
    var ret: [Dictionary<String,String>] = []
    let sparql = personSparql.replacingOccurrences(of: "<name>", with: name)
    print(sparql)
```

```
76      let r = sparqlDbPedia(query: sparql)
77      r.forEach { result in
78          print(result)
79          ret.append(result)
80      }
81      return ret
82  }
83
84  func placeDetail(name: String) -> [Dictionary<String,String>] {
85      var ret: [Dictionary<String,String>] = []
86      let sparql = placeSparql.replacingOccurrences(of: "<name>", with: name)
87      print(sparql)
88      let r = sparqlDbPedia(query: sparql)
89      r.forEach { result in
90          print(result)
91          ret.append(result)
92      }
93      return ret
94  }
95
96  func organizationDetail(name: String) -> [Dictionary<String,String>] {
97      var ret: [Dictionary<String,String>] = []
98      let sparql = organizationSparql.replacingOccurrences(of: "<name>", with: name)
99      print(sparql)
100     let r = sparqlDbPedia(query: sparql)
101     r.forEach { result in
102         print(result)
103         ret.append(result)
104     }
105     return ret
106 }
107
108 public func processEntities(inputString: String) -> [(name: String, type: String, ur\
109 i: String, comment: String)] {
110     let entities = getEntities(text: inputString)
111     var augmentedEntities: [(name: String, type: String, uri: String, comment: Strin\
112 g)] = []
113     for (entityName, entityType) in entities {
114         print("** entityName:", entityName, "entityType:", entityType)
115         if entityType == "PersonalName" {
116             let data = personDetail(name: entityName)
117             for d in data {
118                 augmentedEntities.append((name: entityName, type: entityType,
```

```
                    uri: "<" + d["uri"]! + ">", comment: "<" + d["comment"]! + ">"))
            }
        }
        if entityType == "OrganizationName" {
            let data = organizationDetail(name: entityName)
            for d in data {
                augmentedEntities.append((name: entityName, type: entityType,
                    uri: "<" + d["uri"]! + ">", comment: "<" + d["comment"]! + ">"))
            }
        }
        if entityType == "PlaceName" {
            let data = placeDetail(name: entityName)
            for d in data {
                augmentedEntities.append((name: entityName, type: entityType,
                    uri: "<" + d["uri"]! + ">", comment: "<" + d["comment"]! + ">"))
            }
        }
    }
    return augmentedEntities
}


extension Array where Element: Hashable {
    func uniqueValuesHelper() -> [Element] {
        var addedDict = [Element: Bool]()
        return filter { addedDict.updateValue(true, forKey: $0) == nil }
    }
    mutating func uniqueValues() {
        self = self.uniqueValuesHelper()
    }
}


func getAllRelationships(inputString: String) -> [String] {
    let augmentedEntities = processEntities(inputString: inputString)
    var relationshipTriples: [String] = []
    for ae1 in augmentedEntities {
        for ae2 in augmentedEntities {
            if ae1 != ae2 {
                let er1 = dbpediaGetRelationships(entity1Uri: ae1.uri,
                                                  entity2Uri: ae2.uri)
                relationshipTriples.append(contentsOf: er1)
                let er2 = dbpediaGetRelationships(entity1Uri: ae2.uri,
```

```
                                                    entity2Uri: ae1.uri)
                relationshipTriples.append(contentsOf: er2)
            }
        }
    }
    relationshipTriples.uniqueValues()
    relationshipTriples.sort()
    return relationshipTriples
}
```

AppleBERT

The files in the directory AppleBERT were copied from Apple's example https://developer.apple.com/documentation integration_samples/finding_answers_to_questions_in_a_text_document[73] with a few changes to get returned results in a convenient format for this application. Apple's BERT documentation is excellent and you should review it.

Relationships

The file **Relationships.swift** fetches relationship data for pairs of DBPedia entities. Note that the first SPARQL template has variable slots **<e1>** and **<e2>** that are replaced at runtime with URIs representing the entities that we are searching for relationships between these two entities:

```
// relationships between DBPedia entities

let relSparql =  """
SELECT DISTINCT ?p {<e1> ?p <e2> .FILTER (!regex(str(?p), 'wikiPage', 'i'))} LIMIT 5
"""

public func dbpediaGetRelationships(entity1Uri: String, entity2Uri: String)
                                    -> [String] {
    var ret: [String] = []
    let sparql1 = relSparql.replacingOccurrences(of: "<e1>",
      with: entity1Uri).replacingOccurrences(of: "<e2>",
        with: entity2Uri)
    let r1 = sparqlDbPedia(query: sparql1)
    r1.forEach { result in
        if let relName = result["p"] {
            let rdfStatement = entity1Uri + " <" + relName + "> " + entity2Uri + " ."
            print(rdfStatement)
```

[73] https://developer.apple.com/documentation/coreml/model_integration_samples/finding_answers_to_questions_in_a_text_document

```
            ret.append(rdfStatement)
        }
    }
    let sparql2 = relSparql.replacingOccurrences(of: "<e1>",
        with: entity2Uri).replacingOccurrences(of: "<e2>",
            with: entity1Uri)
    let r2 = sparqlDbPedia(query: sparql2)
    r2.forEach { result in
        if let relName = result["p"] {
            let rdfStatement = entity2Uri + " <" + relName + "> " + entity1Uri + " ."
            print(rdfStatement)
            ret.append(rdfStatement)
        }
    }
    return Array(Set(ret))
}

public func uriToPrintName(_ uri: String) -> String {
    let slashIndex = uri.lastIndex(of: "/")
    if slashIndex == nil { return uri }
    var s = uri[slashIndex!...]
    s = s.dropFirst()
    if s.count > 0 { s.removeLast() }
    return String(s).replacingOccurrences(of: "_", with: " ")
}

public func relationshipsoEnglish(rs: [String]) -> String {
    var lines: [String] = []
    for r in rs {
        let triples = r.split(separator: " ", maxSplits: 3,
            omittingEmptySubsequences: true)
        if triples.count > 2 {
            lines.append(uriToPrintName(String(triples[0])) + " " +
              uriToPrintName(String(triples[1])) + " " +
                uriToPrintName(String(triples[2])))
        } else {
            lines.append(r)
        }
    }
    let linesNoDuplicates = Set(lines)
    return linesNoDuplicates.joined(separator: "\n")
}
```

NLP

The file **NlpWhiteboard** provides high level NLP utility functions for the application:

```
//
//  NlpWhiteboard.swift
//  KGN
//
//  Copyright © 2021 Mark Watson. All rights reserved.
//

public struct NlpWhiteboard {

    var originalText: String = ""
    var people: [String] = []
    var places: [String] = []
    var organizations: [String] = []
    var sparql: String = ""

    init() { }

    mutating func set_text(originalText: String) {
        self.originalText = originalText
        let (people, places, organizations) = getAllEntities(text:  originalText)
        self.people = people; self.places = places; self.organizations = organizatio\
ns
    }

    mutating func query_to_choices(behindTheScenesSparqlText: inout String)
          -> [[[String]]] { // return inner: [comment, uri]
        var ret: Set<[[String]]> = []
        if people.count > 0 {
            for i in 0...(people.count - 1) {
                self.sparql =
                  get_SPARQL_for_finding_URIs_for_PERSON_NAME(nameString: people[i])
                behindTheScenesSparqlText += self.sparql
                let results = sparqlDbPedia(query: self.sparql)
                if results.count > 0 {
                    ret.insert( results.map { [($0["comment"]
                                                ?? ""),
                                                ($0["person_uri"] ?? "")] })
                }
            }
```

```
        }
        if organizations.count > 0 {
            for i in 0...(organizations.count - 1) {
                self.sparql = get_SPARQL_for_finding_URIs_for_ORGANIZATION_NAME(
                    orgString: organizations[i])
                behindTheScenesSparqlText += self.sparql
                let results = sparqlDbPedia(query: self.sparql)
                if results.count > 0 {
                    ret.insert(results.map { [($0["comment"] ??
                      ""), ($0["org_uri"] ?? "")] })
                }
            }
        }
        if places.count > 0 {
            for i in 0...(places.count - 1) {
                self.sparql = get_SPARQL_for_finding_URIs_for_PLACE_NAME(
                    placeString: places[i])
                behindTheScenesSparqlText += self.sparql
                let results = sparqlDbPedia(query: self.sparql)
                if results.count > 0 {
                    ret.insert( results.map { [($0["comment"] ??
                      ""), ($0["place_uri"] ?? "")] })
                }
            }
        }
        //print("\n\n+++++++ ret:\n", ret, "\n\n")
        return Array(ret)
    }
}
```

The file **NLPutils.swift** provides lower level NLP utilities:

```
//  NLPutils.swift
//  KGN
//
//  Copyright © 2021 Mark Watson. All rights reserved.
//

import Foundation
import NaturalLanguage

public func getPersonDescription(personName: String) -> [String] {
    let sparql = get_SPARQL_for_finding_URIs_for_PERSON_NAME(nameString: personName)
```

```
    let results = sparqlDbPedia(query: sparql)
    return [sparql, results.map {
      ($0["comment"] ?? $0["abstract"] ?? "") }.joined(separator: " . ")]
}


public func getPlaceDescription(placeName: String) -> [String] {
    let sparql = get_SPARQL_for_finding_URIs_for_PLACE_NAME(placeString: placeName)
    let results = sparqlDbPedia(query: sparql)
    return [sparql, results.map { ($0["comment"] ??
        $0["abstract"] ?? "") }.joined(separator: " . ")]
}

public func getOrganizationDescription(organizationName: String) -> [String] {
    let sparql = get_SPARQL_for_finding_URIs_for_ORGANIZATION_NAME(
        orgString: organizationName)
    let results = sparqlDbPedia(query: sparql)
    print("=== getOrganizationDescription results =\n", results)
    return [sparql, results.map { ($0["comment"] ?? $0["abstract"] ?? "") }
        .joined(separator: " . ")]
}

let tokenizer = NLTokenizer(unit: .word)
let tagger = NSLinguisticTagger(tagSchemes:[.tokenType, .language, .lexicalClass,
  .nameType, .lemma], options: 0)
let options: NSLinguisticTagger.Options =
    [.omitPunctuation, .omitWhitespace, .joinNames]

let tokenizerOptions: NSLinguisticTagger.Options =
    [.omitPunctuation, .omitWhitespace, .joinNames]

public func getEntities(text: String) -> [(String, String)] {
    var words: [(String, String)] = []
    tagger.string = text
    let range = NSRange(location: 0, length: text.utf16.count)
    tagger.enumerateTags(in: range, unit: .word,
        scheme: .nameType, options: options) { tag, tokenRange, stop in
        let word = (text as NSString).substring(with: tokenRange)
        let tagType = tag?.rawValue ?? "unkown"
        if tagType != "unkown" && tagType != "OtherWord" {
            words.append((word, tagType))
        }
    }
```

```
55      return words
56  }
57
58  public func tokenizeText(text: String) -> [String] {
59      var tokens: [String] = []
60      tokenizer.string = text
61      tokenizer.enumerateTokens(in: text.startIndex..<text.endIndex) { tokenRange, _ in
62          tokens.append(String(text[tokenRange]))
63          return true
64      }
65      return tokens
66  }
67
68  let entityTagger = NLTagger(tagSchemes: [.nameType])
69  let entityOptions: NLTagger.Options = [.omitPunctuation, .omitWhitespace, .joinNames]
70  let entityTagTypess: [NLTag] = [.personalName, .placeName, .organizationName]
71
72  public func getAllEntities(text: String) -> ([String],[String],[String]) {
73      var words: [(String, String)] = []
74      var people: [String] = []
75      var places: [String] = []
76      var organizations: [String] = []
77      entityTagger.string = text
78      entityTagger.enumerateTags(in: text.startIndex..<text.endIndex, unit: .word,
79          scheme: .nameType, options: entityOptions) { tag, tokenRange in
80          if let tag = tag, entityTagTypess.contains(tag) {
81              let word = String(text[tokenRange])
82              if tag.rawValue == "PersonalName" {
83                  people.append(word)
84              } else if tag.rawValue == "PlaceName" {
85                  places.append(word)
86              } else if tag.rawValue == "OrganizationName" {
87                  organizations.append(word)
88              } else {
89                  print("\nERROR: unkown entity type: |\(tag.rawValue)|")
90              }
91              words.append((word, tag.rawValue))
92          }
93          return true
94      }
95      return (people, places, organizations)
96  }
97
```

```
98  func splitLongStrings(_ s: String, limit: Int) -> String {
99      var ret: [String] = []
100     let tokens = s.split(separator: " ")
101     var subLine = ""
102     for token in tokens {
103         if subLine.count > limit {
104             ret.append(subLine)
105             subLine = ""
106         } else {
107             subLine = subLine + " " + token
108         }
109     }
110     if subLine.count > 0 {
111         ret.append(subLine)
112     }
113     return ret.joined(separator: "\n")
114 }
```

Views

This is not a book about SwiftUI programming, and indeed I expect many of you dear readers know much more about UI development with SwiftUI than I do. I am not going to list the four view files:

- MainView.swift
- QueryView.swift
- AboutView.swift
- InfoView.swift

Main KGN

The top level app code in the file **KGNApp.swift** is fairly simple. I hardcoded the window size for macOS and the window sizes for running this example on iPadOS or iOS are commented out:

```
import SwiftUI

@main
struct KGNApp: App {
    var body: some Scene {
        WindowGroup {
          MainView()
            .frame(width: 1200, height: 770)    // << here !!
            //.frame(width: 660, height: 770)    // << here !!
            //..frame(width: 500, height: 800)    // << here !!
        }
    }
}
```

I was impressed by the SwiftUI framework. Applications are fairly portable across macOS, iOS, and iPadOS. I am not a UI developer by profession (as this application shows) but I enjoyed learning just enough about SwiftUI to write this example application.

Part 5: Apple Intelligence

Developers Can Now Weave Apple Intelligence Directly Into Their Apps

Dear reader, Apple has opened up its new "Apple Intelligence" system to third-party developers, providing a suite of tools and APIs to create more personalized and powerful applications. Developers can primarily leverage this technology through the new Foundation Models framework, which offers direct access to Apple's on-device generative models. This allows for the integration of sophisticated features like text summarization, content creation, and message rewriting directly within an app's interface, all while processing user data locally on the device to ensure privacy.

Furthermore, developers can utilize an expanded App Intents framework to make their application's content and functionalities accessible to system-wide services like Siri and Shortcuts. This deeper integration allows users to interact with apps using natural language and create complex, multi-app workflows. For instance, a user could ask Siri to "find all photos of my dog from last summer and create a collage in my favorite editing app," and the system, through App Intents, would understand and execute this command. The Image Playground API also allows for the seamless inclusion of image generation capabilities, enabling users to create unique visuals within the context of the app they are using.

Key Advantages for Developers:

- Enhanced User Experience: By integrating features like personalized content suggestions, intelligent summarization, and natural language interaction, developers can create more intuitive and engaging applications that anticipate user needs.
- On-Device Processing for Privacy and Performance: A significant advantage is the on-device nature of Apple Intelligence. This approach enhances user privacy and security by keeping personal data on the device. It also improves app performance and responsiveness by reducing reliance on cloud-based processing.
- Access to Powerful Generative Models: Developers gain access to Apple's sophisticated generative models without the need to build and train their own. This allows for the rapid implementation of advanced AI features that were previously complex and resource-intensive to develop.
- Deeper System Integration: The enhanced App Intents framework provides a new level of integration with core iOS, iPadOS, and macOS features. This allows apps to become more discoverable and useful within the broader Apple ecosystem, extending their functionality beyond the confines of the app itself.

Using Apple Intelligence's Default System Model To Build a Chat Command Line Tool

Here we create a simple command line chat tool. The LLM specific code can also be used in iOS, iPadOS, and macOS applications. This chat tool will use the local system LLM for simple queries and will transparently call a more capable model on Apple's servers in a secure and privacy preserving sandbox.

The following Swift package manifest defines an executable command-line program named **chattool** designed to run on macOS. The tool relies on Apple's swift-argument-parser package to process command-line arguments.

Package.swift:

```
// swift-tools-version: 6.2
import PackageDescription

let package = Package(
    name: "ChatTool",
    platforms: [.macOS(.v26)],   // macOS 15/16 is fine too
    products: [
        .executable(name: "chattool", targets: ["ChatTool"])
    ],
    dependencies: [
        // □ Apple's official argument-parsing package
        .package(url: "https://github.com/apple/swift-argument-parser.git",
                 from: "1.3.0")
    ],
    targets: [
        .executableTarget(
            name: "ChatTool",
            // expose the ArgumentParser product to the target
            dependencies: [
                .product(name: "ArgumentParser", package: "swift-argument-parser")
                // FoundationModels is an Apple framework, no SPM entry needed
            ],
            path: "Sources/ChatTool"          // adjust if your path differs
```

```
        )
    ]
)
```

ChatTool.swift:

This Swift code implements a command-line chat interface that interacts directly with Apple's native FoundationModels framework. Upon launch, it verifies the system's default language model is available, initializes a LanguageModelSession with a hard-coded system prompt and temperature, and then enters a read-evaluate-print loop. The program continuously accepts user input from the console and sends it to the language model for processing until the user quits.

The core of the implementation leverages modern Swift concurrency (async/await) to handle the model's output as an asynchronous stream. For each user prompt, it iterates through the text fragments as they are generated by session.streamResponse, writing only the new characters to standard output to create a real-time, typewriter-like effect. To enhance usability, it uses a DispatchSource to set up a signal handler for SIGINT (Ctrl+C), allowing a user to cancel the current in-progress stream from the model without terminating the entire chat application.

```
import Foundation
import FoundationModels
import Dispatch

@main
struct ChatCLI {
    static func main() async throws {
        // Hard-coded defaults
        let temperature  = 0.2
        let sysPrompt    = "You are a helpful assistant."

        // Verify model
        let model = SystemLanguageModel.default
        guard model.isAvailable else {
            throw RuntimeError("Model unavailable: \(model.availability)")
        }

        let session = LanguageModelSession(instructions: sysPrompt)
        print("Temperature: \(temperature)")
        print("System Prompt: \(sysPrompt)")
        let options = GenerationOptions(temperature: temperature)

        print("Apple-Intelligence chat (streaming, T=0.2). Type /quit to exit.\n")

        while true {
```

```
            print("Enter your message: ", terminator: "")
            guard let prompt = readLine(strippingNewline: true) else { break }
            if prompt.isEmpty || prompt == "/quit" { break }

            var previous = ""       // text already printed

            let task = Task {
                for try await part in session.streamResponse(to: prompt, options: op\
tions) {
                    let delta = part.dropFirst(previous.count) // new characters only
                    if !delta.isEmpty {
                        FileHandle.standardOutput.write(Data(delta.utf8))
                        fflush(stdout)
                        previous = part
                    }
                }
                print() // newline when complete
            }

            // ^C cancels the streaming task
            signal(SIGINT, SIG_IGN)
            let sigSrc = DispatchSource.makeSignalSource(signal: SIGINT, queue: .mai\
n)
            sigSrc.setEventHandler { task.cancel() }
            sigSrc.resume()
            defer { sigSrc.cancel() }

            _ = try await task.value
        }
    }
}

/// Simple error wrapper
struct RuntimeError: Error, CustomStringConvertible {
    let description: String
    init(_ msg: String) { description = msg }
}
```

Here are the first few lines of output given the prompt *Describe the math for calculating the orbit of Jupiter, then write a very short design for a Python script*:

```
$ swift run
[1/1] Planning build
Building for debugging...
[1/1] Write swift-version-39B54973F684ADAB.txt
Build of product 'chattool' complete! (0.16s)
Temperature: 0.2
System Prompt: You are a helpful assistant.
Apple-Intelligence chat (streaming, T=0.2). Type /quit to exit.

Enter your message: Describe the math for calculating the orbit of Jupiter, then wri\
te a very short design for a Python script
Calculating the orbit of Jupiter involves solving Kepler's laws of planetary motion,\
 which describe the elliptical orbits of planets around the Sun. The key equations i\
nvolve gravitational forces and conservation laws. Here's a brief overview of the ma\
th involved:

 ### Key Concepts:

1. **Kepler's Laws:**
   - **First Law (Law of Ellipses):** Planets move along ellipses with the Sun at on\
e focus.
   - **Second Law (Law of Equal Areas):** A line segment joining a planet and the Su\
n sweeps out equal areas during equal intervals of time.
   - **Third Law (Law of Harmonies):** The square of the orbital period (\(T\)) is p\
roportional to the cube of the semi-major axis (\(a\)): \(T^2 = \frac{4\pi^2}{GM}a^3\
\).

2. **Gravitational Force:**
   - The gravitational force between two masses (\(m_1\) and \(m_2\)) is given by Ne\
wton's law: \(F = G \frac{m_1 m_2}{r^2}\), where \(G\) is the gravitational constant\
 and \(r\) is the distance between centers.

3. **Centripetal Force:**
   - For circular orbits, centripetal force equals gravitational force: \(F = \frac{\
mv^2}{r}\).
```

Using Apple Intelligence's Default System Model To Build a Coding Assistant Command Line Tool

This tool looks in the current directory and all subdirectories for source code files and describes them and then enters a chat loop for talking about the code.

Package.swift:

```
// swift-tools-version: 6.2
// The swift-tools-version declares the minimum version of Swift required to build t\
his package.

import PackageDescription

let package = Package(
    name: "CodingCLI",

    // 1 Tell SwiftPM we require at least macOS 12 so
    //    `Task.value`, async/await, and FoundationModels are available.
    platforms: [
        .macOS(.v26)
    ],

    products: [
        .executable(name: "CodingCLI", targets: ["CodingCLI"])
    ],

    targets: [
        .executableTarget(
            name: "CodingCLI",

            // 2 Link the system framework that ships with Xcode 17+
            //    (no external dependency required).
            linkerSettings: [
                .linkedFramework("FoundationModels")
            ]
        )
```

```
    ]
)
```

CodingCLI.swift:

```
import Foundation
import FoundationModels
import Dispatch

@main
struct CodingCLI {
    static func main() async throws {
        // ---- 1. Gather candidate source files ----
        let exts = ["swift", "py", "lisp"]
        var blobs: [String] = []

        let enumerator = FileManager.default.enumerator(atPath: ".")!

        while let path = enumerator.nextObject() as? String { // avoids @noasync
            guard let ext = path.split(separator: ".").last,
                  exts.contains(ext.lowercased()) else { continue }

            if let data = FileManager.default.contents(atPath: path),
               data.count < 8 * 1024 {                          // keep size filter
                let text = String(decoding: data, as: UTF8.self) // non-optional
                blobs.append("### \(path) ###\n\(text)")
            }
        }

        let doc      = blobs.joined(separator: "\n")
        let summary  = try await Self.summarize(doc)
        print("\n=== Project Summary ===\n\(summary)\n")

        // ---- 2. Start interactive chat loop ----
        let session  = LanguageModelSession(instructions:
                                            "You are a helpful assistant.")
        let options  = GenerationOptions(temperature: 0.2)
        print("Apple-Intelligence chat (streaming, T=0.2).  Type /quit to exit.\n")

        while let prompt = readLine(strippingNewline: true) {
            if prompt.isEmpty || prompt == "/quit" { break }

            var printed = ""
```

```
            let task = Task {
                for try await part in session.streamResponse(to: prompt,
                                                             options: options) {
                    let delta = part.dropFirst(printed.count)
                    if !delta.isEmpty {
                        FileHandle.standardOutput.write(Data(delta.utf8))
                        fflush(stdout)
                        printed = part
                    }
                }
                print()
            }

            signal(SIGINT, SIG_IGN)
            let sig = DispatchSource.makeSignalSource(signal: SIGINT, queue: .main)
            sig.setEventHandler { task.cancel() }
            sig.resume()
            defer { sig.cancel() }

            _ = try await task.value
        }
    }

    // ---- 3. Helper: summarize all code ----
    static func summarize(_ text: String) async throws -> String {
        let session = LanguageModelSession(
            instructions: """
            Summarize the following multi-file project. \
            For each file give one bullet explaining its role, then \
            a two-sentence overall description.
            """
        )
        let prompt = text.prefix(24 * 1024)                // safety window
        let resp   = try await
                     session.respond(to: String(prompt),
                                     options: GenerationOptions(temperature: 0))
        return resp.content                            // unwrap Response<String>
    }
}
```

Here is the output for running this tool in its own source directory:

```
$ swift run
Building for debugging...
[8/8] Applying CodingCLI
Build of product 'CodingCLI' complete! (3.23s)

 === Project Summary ===
 ### test.py
- **Role:** This script interacts with Groq to perform a chat completion task.
- **Description:** It sets up a chat session using Groq's API, sends a specific mess\
age, and prints the response, showcasing how to utilize Groq for conversational AI t\
asks.

 ### Package.swift
- **Role:** Defines the Swift package configuration for the CodingCLI project.
- **Description:** This file specifies the project's platform requirements, defines \
the executable product, and outlines the executable target with necessary dependenci\
es.

 ### Sources/CodingCLI/CodingCLI.swift
- **Role:** Serves as the entry point for the CodingCLI application, handling file s\
ummarization and chat interaction.
- **Description:** It processes source files to generate a summary, and manages an i\
nteractive chat loop using a language model, demonstrating integration of summarizat\
ion and conversational AI within a Swift package.

Apple-Intelligence chat (streaming, T=0.2).  Type /quit to exit.
```

Book Wrap Up

I hope that you dear reader enjoyed this short book. While I enjoy programming in Swift and appreciate how well Apple has integrated machine learning capabilities in their iOS/iPadOS/macOS ecosystems, I still find myself writing most of my experimental code in Lisp languages and using Python for deep learning experiments and projects. That said, I am very happy that I have done the work to add Swift, CoreML, and SwiftUI to my personal programming tool belt.

I usually update my eBooks so if there is some topic or application domain that you would like added to future versions of this book, then please let me know. My email address is markw <at> markwatson <dot> com.