Introduction

- Importance of the Choice of the topic;

- Statement of the problem;

- Analytical Review of the state of the issue/ Literature Review;

Translation between the native languages of mobile development.

As we all know we use our mobiles everyday and for almost everything now, but we all don’t have the same phone, I’m mostly talking about the two main operating systems that are in the market, that utilizes both their own proprietary language. As the two main players of this space we encounter Google with Android and Apple with iOS, the languages being Kotling and Swift.

There are some solutions out there, like the flutter sdk with dart language you can create apps that can run in both operating system (you can see an example in my github: [leofanky/convo (github.com)](https://github.com/leofanky/convo) ) because the final code is compile in native code but the functionality is really limited still and is not fully developed yet as is still in beta stage in many aspects, there are some other similar alternative to flutter but they all work with the same logic.

I found some project online trying to do a translation as an example: Gryphon (https://github.com/vinivendra/Gryphon), but it is only a preview and they don’t use ML or any other kind of new technologies but is more like a dictionary, and the main issue is that is only a one way translation.

Main issue in the market is for small to medium companies that face the challenge of choosing a platform as there aren’t many developers that code in both and furthermore they are really requested and with a really high salary. So the idea would be to take your native code being Kotlin or Swift and translating to the other, using a simple text box where you input your code and it outputs the code in the other language.

Main

-Description and justification of the Method;

-Presentation and analysis of the result;

-Analytical Comparison with already known result;

**Why choose a ML algorithm for code translation?**

Not so long ago the public had no idea computers could generate code by themself until we started seeing big Tech companies that needed pipelines for automated testing and for code snippet generation for speeding up the process of coding and testing.

At the same time we see a big divide, especially in the mobile industry, there are the two big players that we all know and use at least one of them everyday:

* Google/Android
* Apple/iOS

Both of them are already using a lot of ML to analyze behaviour and better apps and the OS, one of the biggest parts being voice assistance on every device now a days, but this companies don’t only use this logic of “AI>human”, they are using it for Database management, even management, human resource.

So I thought why not implement Machine Learning on the creation and implementation of apps, helping many small companies that can’t pay many developers, but also helping autonomous developers to make their codes available easier on other platforms.

So here is where my idea of using NLP (Natural Language Processing), that is one of the biggest parts of ML, and AI in general and really good at translating natural language, maybe I could implement something to translate one code to another.

As explained previously iOS language (Swift) and Android language (kotlin) are high level languages, what does this mean? It means that they are easier to read, write, and maintain because they have a higher level of abstraction and as such they are human readable and have a similar structure to a human understandable language, meaning that programs written in a high-level language must be translated into machine language by a compiler or interpreter.

So Logically I started trying to code some NLP model and train it on my machine, as soon as I started looking to a database to train on I realize that of course there wouldn’t be any as I came up with this idea, google it and nothing came up, nobody had ever tried this. I started to try to make one mine but then I realized already on my first training it was too slow, so I decided to look for a model pre-trained where I could tweak it and train it more on my small dataset. The first thing that came up was OpenAI, an it’s model GPT-3, when I applied for the api key as I was intrigued what I could, while waiting for they key to be sent I started looking for example on the internet, it seemed like there was a vague reference to making use on the model to translate Ruby to Python code:

(https://www.notion.so/Code-Translation-416ad93f325a43c28f63ebd74a9158d3)

But this was just for the playground they allow you to use, then I started looking out on how to deploy my playground and how to tweak the model for my scope. I decided then that I would build a React + Flask webapp that would allow devs to use it on the internet or even with their out keys deployed on their own machine.

**GPT-3 Defined by wikipedia**:

Generative Pre-trained Transformer 3 (GPT-3) is an autoregressive language model that uses deep learning to produce human-like text. It is the third-generation language prediction model in the GPT-n series (and the successor to GPT-2) created by OpenAI, a San Francisco-based artificial intelligence research laboratory. GPT-3's full version has a capacity of 175 billion machine learning parameters. GPT-3, which was introduced in May 2020, and was in beta testing as of July 2020, is part of a trend in natural language processing (NLP) systems of pre-trained language representations. Before the release of GPT-3, the largest language model was Microsoft's Turing NLG, introduced in February 2020, with a capacity of 17 billion parameters or less than a tenth of GPT-3s.

So this is not really useful to fully understand what GPT-3 is and why it works so well, and gives us only a vague definition of what is really happening behind the curtains, for that we need to read and explain the Paper of GPT-3: Language Models are Few-Shot Learners.

GPT-3

Language Models are Few-Shot Learners

First we need to define a few things to explain this paper, so here are some definitions for you.

**Tokens:**

Tokens are pieces of words, subwords, in the NLP world it is common to leverage tokens to handle unknown words and improve efficiency.

Example: the word “lower” gets broken up into the tokens “low”, “er”, and a shorter and common word like “pear” is a single token. On average one token is roughly 4 characters talking about the English Language.

**Language model:**

by simple words is a model that tries to predict on what comes next. It can generate Language in a probabilistic way.

**Transformer Model:**

Transformers are designed to handle sequential data, like natural language, for tasks such as translation and text summarization however Transformers do not require that the sequential data be processed in order. For example, if the input data is a natural language sentence, the Transformer does not need to process the beginning of it before the end. Due to this feature, the Transformer allows for much more parallelization reducing training times.

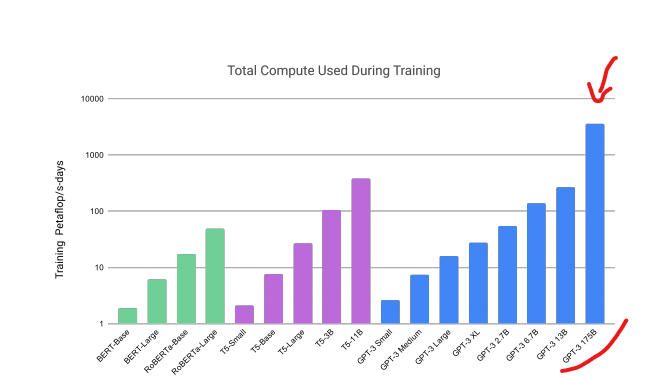
Now that we defined thisa 3 things we can start explaining what is GPT-3 and how it is trying to accomplish these things.

GPT-3 has a huge dataset that is trained on, way bigger than any model creator previously even from the likes of Amazon, Google or Microsoft.

**GPT-3 training dataset**:

* Common Crawl (filtered)
* WebText2
* Books1
* Books2
* Wikipedia

Common Crawl being a internet crawl, meaning most websites on the surface web are in this dataset, the total sum of tokens on all of them is 499 billion tokens with 82%(410 billion tokens) being on the Common Crawl



This is how much computational power they’ve used during the training of the model.

**What if we want to make a Transformer Model into a Language model**: the input would be the context (meaning the sentence), the model creates several layer of attention mechanism (attention mechanism is basically a way where information is routes in between the different tokens) and as it goes up the layers the information is routed around and the model can make various inferences and at the end the model is suppose to come up with the next word of the sentence. This kind of model is an autoregressive language model meaning that it can only go from left to right for the "autocompletion".

After a model choice, in the standard approach to creating these models you’d need fine tuning, but GPT uses a different way of doing it. They are using a Few-Shot Learners approach, for contrast:

* **zero-shot** the model would predict the answer given only a natural language description of the task and no gradient update are performed, example below:

Translate English to French: ⇐ task description

Cheese ⇒ \_\_\_\_\_\_\_\_\_\_\_\_\_ ⇐ prompt

The reality is that here you are just hoping that the model has seen enough times this structure that it knows what it is meant to do.

* **One-shot** in addition to the task description, the model sees a single example of the task and no gradient update are performed, example below:

Translate English to French: ⇐ task description

Sea otter ⇒ loutre de mer ⇐ example

Cheese ⇒ \_\_\_\_\_\_\_\_\_\_\_\_\_ ⇐ prompt

The advantage is that we need only one model and not needing to train a bunch of models for every single task, as we are providing a sort of training data with our example.

* **Few-shot** similar to one-shot but the models see a few examples of the task. Also here not gradient update are performed , example below:

Translate English to French: ⇐ task description

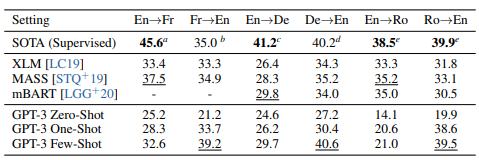
Sea otter ⇒ loutre de mer ⇐ example

Peppermint ⇒ menthe poivrée ⇐ example

Plush giraffe ⇒girafe peluche ⇐ example

Cheese ⇒\_\_\_\_\_\_\_\_\_\_\_\_\_ ⇐ prompt

As this is the best solution for now only reason being is because of the enormity of the data that the current model of GPT is trained on, as this is not a viable solution for any other small model.



These are the results they got for translation, as we can see it even outperforms State Of The Art (supervised) model, in language of course, so this is the main part of why I’m using it myself for translation.

You can see more and the result in the paper, this is just a short explanation on how the model works, and in the ambit of GPT these examples are called “priming”. (paper: <https://arxiv.org/abs/2005.14165>).

Main differences between kotlin and swift

**Syntactical Differences** that you can see below:



Optional and Nullable Types

Both Swift and Kotlin have put effort into ensuring devs don’t accidentally end up with the null pointer exceptions, both use a kind of wrapper type that lets you safely access the underlying values: Swift calls this type **Optionals**, Kotlin calls them nullables. All you need to do is set a **?** after the variable type.

Kotlin nullable string when used in action prints the same string as a non-nullable type. Different from Swift, it doesn’t wrap itself in an optional wrapper. This difference is caused by the way Optionals are implemented in Swift as they’re essentially an **enum**.

Kotlin nullables contain a null type checker, which the compiler uses when the bytecode is generated to handle the null references.

In both a **?** needs to be placed after the object for querying properties or calling functions. In Swift, this is called optional chaining, and in Kotlin, it’s called safe calls.

Swift has a nil-coalescing operator, **??,** that safely unwrap a value if it exists or provides a default value when empty. In Kotlin, the same can be done using a ?: Elvis operator.

To handle Optionals properly and apply some additional operations on the unwrapped values, should use a null checker or smart cast ( **is** or **as**) in Kotlin. Swift, on the other hand, provides us with **if let** and **guard let** statements that are a lot more readable.

There’s no similar optional binding syntax for Kotlin, it does provide a **let()** lambda function that executes the action only when the value is non-nullable. But this is different from the **if let** and **guard let** approaches since it doesn’t let you access the local variable in the scope.

While it is not recommended to use force unwrapping or not-null assertion, the syntax for doing it is **!** for Swift and **!!** for Kotlin.

Implicit wrapping in Swift indicates to the compiler to ignore the safe unwrap since the property would always have a value available. In Kotlin, the same is done by using the **lateinit** keyword in the variable declaration.

Both Kotlin and Swift allow safe casting using **as?** .

Control Flows

**if-else**, **for**, and **switch** are the basic control flow statements available in most languages.

In Swift, the parentheses around the condition aren’t necessary, but setting curly braces around the if-else condition is mandatory. Kotlin does just the opposite, similar to Java and lets you get away with curly braces in one-liner conditions, thereby posing ambiguity due to the dangling else problem.

Kotlin and Swift both have differed from C-style for loops and instead give more emphasis to **for in** loops. There is the ability to create **for** loops using the range operator.

The **until** operator in Kotlin is analogous to **..<** in Swift. Both exclude the last value. There’s an alternative way to increment or decrement values in the for loop iterations as well.

Swift presents us with a stride operator, which has two variants:

**stride(from:to:by:)**  (excludes the to value)

**stride(from:through:by:)** (includes the through value)

Kotlin provides us with **step** to control the increment and **downTo** for creating steps in reverse order.

**Switch** statement in Swift is equal to **when** in Kotlin. The Swift **switch** needs to be exhaustive, it isn’t necessary with Kotlin’s statement **when** (unless the statement is returning a value).

Both statements are incredibly powerful and offer pattern matching, multiple cases, and typecasting, for some examples. In Swift using a **case let** or range operators, you can do a partial match. Kotlin provides us with **in** (range of argument) and **is** (smart casting) operators while not letting you pass arguments in the **when** statement.

Classes, Functions, Constructors, and Inheritance

Basic declaration of classes and functions in both languages is almost the same, but there are some subtle differences.

The Kotlin classes, functions, and properties defined are **final** by default. Meaning that they can’t be inherited, to allow subclassing or overriding, we need to declare them as **open class** and **open fun**.

Swift functions support internal and external parameter names, improving the function’s readability. A variable number of arguments in Kotlin can be specified using:

**fun sum(vararg numbers: Int)**

and in Swift, the following signature is used:

**func sum(\_ numbers: Int...)**

Constructors are implemented in different ways. Kotlin lets you declare the primary constructor in the class header, with an option to set default values, secondary constructors are declared using the **constructor** keyword in the class body. If the class contains a primary constructor, the secondary constructor must refer to it using **this**. There’s an **init** block that runs whenever the class is initialized.

Constructor arguments are **val** by default, unless you specify them as **var**. As for subclasses in Kotlin, in case a primary constructor isn’t defined, the secondary constructor must use the **super** keyword to call the base class or delegate to another constructor that does that using **this**.

In Swift, the **init** block represents the primary constructor. It is a designated initializer that’s responsible for setting up all the properties of that class, Swift has a secondary initializer known as a **convenience** initializer, designated to the primary one by letting you assign default values to some properties.

Swift includes **required** and automatic-initializer inheritance. The **required** keyword set on any initializer it enforces you to implement that initializer in subclasses. Automatic-initializer inheritance follows two rules:

* Don’t define any designated initializers in your subclass — they’ll be inherited automatically
* Implement all the designated initializers of the base class. All the convenience initializers will be inherited implicitly.

Structs and Data Classes

Kotlin’s primary constructor syntax makes it easy to create a data class for storing data. But Kotlin doesn’t have a way to create a custom type for passing data by value, like structures in Swift.

Closures

Lambda expressions aren’t just functions without a name. Closures are a type of lambda that lets you capture values outside of the scope it’s defined in. The return type of a closure is specified after **->** in both Swift and Kotlin. Kotlin provides an **it** parameter for single-value closures, and the **->** can be escaped. Swift, on the contrary, lets you use **$0**, **$1**, **$2**, and so on.

While the Swift closures can be defined as **escaping** and **non-escaping**, Kotlin closures are **escaping** by default. This has to do with the memory-management systems of the two languages: Swift uses **Automatic Reference Counting** and **Kotlin uses the Garbage Collector**.

Extensions

Extensions in Swift lets you add extra functionality to classes, structs, enums, and protocols without actually extending them. In Kotlin, extensions are more like static methods of Java. Kotlin provides **get()** and **set(value)** methods by default for all property declarations. Swift, on the other hand, makes use of a more flexible computed property syntax with **get{}** and **set{}** and the property observers **didSet** and **willSet**, which are triggered when the value is changed.

Interfaces and Protocols

Interfaces in Kotlin are known as protocols in Swift. Both are responsible for defining a set of functions or properties the class can conform to. The difference is Kotlin allows us to set default implementations in the interface. As a workaround for Swift, you can use extensions on the protocols.

Also for generics, Swift lets you define a **associatedType** in the protocol; whereas for Kotlin, you can use the generic type **T** in the definition.

Enumerations

Enumerations in Kotlin are a class that holds a bunch of constants with a custom data type. On the other hand, in Swift, **enums** are first-class citizens that let us specify the associated value of any type with every case. Kotlin **Enums**, on the other hand, can have only one object per type.

Now, Kotlin does have a workaround in the form of sealed classes, which let you create instances with different types. But again, sealed classes are a reference type which entirely replicates the Swift **enums** value type.

Lazy

At last we have **lazy** in Swift that allows you to defer the initialization of the property till first access.With Kotlin can be done the same using the delegated properties **by** feature. While with Swift, **lazy** is a keyword that’s applicable only to mutable properties, the **by lazy** accepts a lambda function and is applicable to only immutable properties (it can’t be changed once set).

So here are most of the differences from both languages, from that I took inspiration and created a few “prompts”, of as explained example where GPT-3 could look on how to do, this is what I came up with:

**Hello World & Variables And Constants**

**Swift**:

var myVariable = 42

myVariable = 50

let myConstant = 42

let oranges = 5

print("Hello, world!")

let fruitSummary = "I have \(myConstant + oranges)" + "pieces of fruit."

**Kotlin**:

var myVariable = 42

myVariable = 50

val myConstant = 42

println("Hello, world!")

val oranges = 5

val fruitSummary = "I have ${myConstant + oranges}" + "pieces of fruit."

**Type Conversion**

**Swift**:

let label = "The width is "

let width = 94

let widthLabel = label + String(width)

**Kotlin**:

val label = "The width is "

val width = 94

val widthLabel = label + width

**Range Operator & Collections(Arrays) &**

**Empty Collections**

**Swift**:

let names = ["Anna", "Alex", "Brian", "Jack"]

let count = names.count

for i in 0..<count {

print("Person \(i + 1) is called \(names[i])")}

names[1] = "Leo"

let emptyArray = [String]()

let emptyDictionary = [String: Float]()

**Kotlin**:

val names = arrayOf("Anna", "Alex", "Brian", "Jack")

val count = names.count()

for (i in 0..count - 1) {

println("Person ${i + 1} is called ${names[i]}")}

names[1] = "Leo"

val emptyArray = arrayOf<String>()

val emptyMap = mapOf<String, Float>()

**Tuple Return**

**Swift**:

func getGasPrices() -> (Double, Double, Double) {

return (3.59, 3.69, 3.79)

}

**Kotlin**:

data class GasPrices(val a: Double, val b: Double,

val c: Double)

fun getGasPrices() = GasPrices(3.59, 3.69, 3.79)

**Higher Order function (Map) & Sort**

**Swift**:

let numbers = [20, 19, 7, 12]

numbers.map { 3 \* $0 }

var mutableArray = [1, 5, 3, 12, 2]

mutableArray.sort()

**Kotlin**:

val numbers = listOf(20, 19, 7, 12)

numbers.map { 3 \* it }

listOf(1, 5, 3, 12, 2).sorted()

Classes

**Swift**:

class Shape {

var numberOfSides = 0

func simpleDescription() -> String {

return "A shape with \(numberOfSides) sides."

}

}

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

**Kotlin**:

class Shape {

var numberOfSides = 0

fun simpleDescription() =

"A shape with $numberOfSides sides."

}

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

**Extensions**

**Swift**:

extension Double {

var km: Double { return self \* 1\_000.0 }

var m: Double { return self }

var cm: Double { return self / 100.0 }

var mm: Double { return self / 1\_000.0 }

var ft: Double { return self / 3.28084 }

}

let oneInch = 25.4.mm

print("One inch is \(oneInch) meters")

let threeFeet = 3.ft

print("Three feet is \(threeFeet) meters")

**Kotlin**:

val Double.km: Double get() = this \* 1000

val Double.m: Double get() = this

val Double.cm: Double get() = this / 100

val Double.mm: Double get() = this / 1000

val Double.ft: Double get() = this / 3.28084

val oneInch = 25.4.mm

println("One inch is $oneInch meters")

val threeFeet = 3.0.ft

println("Three feet is $threeFeet meters")

**Interface & Protocol & Extension**

**Swift**:

protocol MyProtocol {

init(parameter: Int)

var myVariable: Int { get set }

var myReadOnlyProperty: Int { get }

func myMethod()

func myMethodWithBody()

}

extension MyProtocol {

func myMethodWithBody() {

// implementation goes here

}

}

**Kotlin**:

interface MyInterface {

var myVariable: Int

val myReadOnlyProperty: Int

fun myMethod()

fun myMethodWithBody() {

// implementation goes here

}

}

**Functions**

**Swift**:

func greet(\_ name: String,\_ day: String) -> String {

return "Hello \(name), today is \(day)."

}

greet("Bob", "Tuesday")

**Kotlin**:

fun greet(name: String, day: String): String {

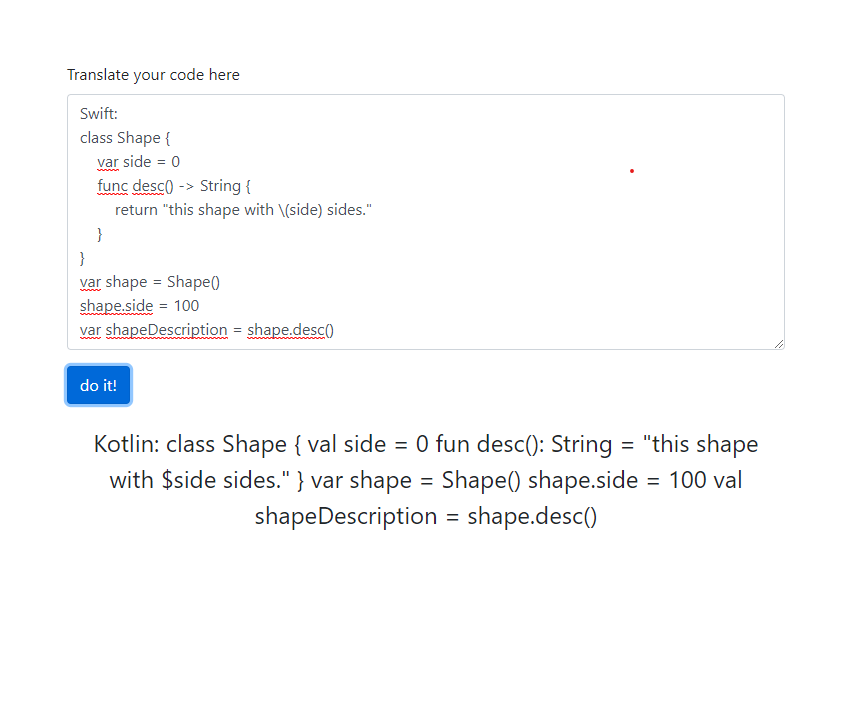
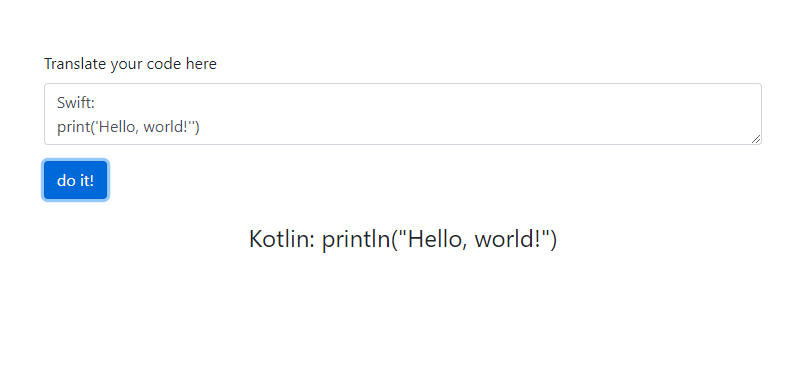
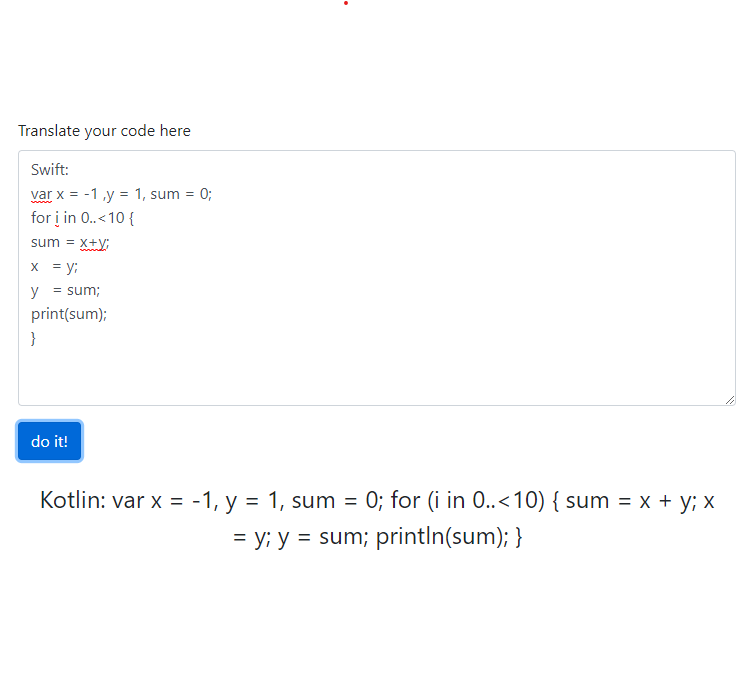
return "Hello $name, today is $day."

}

greet("Bob", "Tuesday")

Conclusion & recommendations:

So the conclusion is that if the model keeps getting bigger and bigger as OpenAI is saying, future models are going to be exponentially better at it and with always more and more “priming” being done is just a question of how much time until it can translate the entire app even proprietary code.

Here are some working examples of the React+Flask WebApp:

Code:

pip install -r requirements.txt

set OPENAI\_CONFIG=Code\_translation/openai.cfg

yarn install

The start the web app with:

Python code\_trasnlation/run\_code\_trasnlation.py