TubeNap:
take a nap while commuting

Author: Filippo Simone

Superviser: Dr Keith Leonard Mannock

MSc Computer Science

Project report

Department of Computer Science and Information Systems, Birkbeck College, University of London 2019

Abstract

The following project report will describe the development process of an iOS application by focusing on the knowledge and the tools required for its implementation. The presented program allows the user to plan his journey on London's tube network, and subsequently handles the process of notifying the user when the trip has ended, by using a combination of machine learning algorithm for sound classification, and information retrieval via the TfL Open Data API, for extracting meaningful information, such as estimated duration and stations along the predefined journey path.

The application was developed following a bootstrap method in which each component was first implemented as a separate feature and then integrated into the application. Each function was coded following TDD practices. The application was conceived to be minimal in its user interface and intuitively easy to understand. The final prototype is a fully functional system that can be easily extended to include other means of transportation and cities in future updates.

Table of Contents

1. Introduction *1*

1.1 Case description 1

1.2 Available approaches 1

1.3 Roadmap 2

2. Structure of an iOS app *2*

2.1 iOS SDK and Swift programming language 2

2.2 MVC design pattern 3

2.1.1 MVVM and MVP instead of MVC 4

2.2.2 Communication 4

2.2 UIKit 5

2.2.3 Interface Builder 5

2.2.2 IBActions and IBOutlets 6

2.3 Segue 6

3. Requirements *7*

3.1 Data 7

3.2. Journey Plan 8

4. Development *8*

4.1 Persistent storage 8

4.2 TfL data request 12

4.3 Sound Classification 14

4.3.1 Improvements to the model 17

4.3.2 Exporting the model to Xcode 18

5. Model-View-Controller Implementation *21*

5.1 Model implementation 21

5.2 Controller implementation 21

5.3 View implementation 27

6. Evaluation *29*

7. Conclusion *32*

8. References *33*

1. Introduction

1.1 Case description

With commuting being a central part of the daily routine of a Londoner, on average residents in London complete two trips every day with an average commute time of 30 minutes. This number represents a significant share of our time allocation during a day and considering it over the working-age period of an individual, this time could be used for productive activities that positively affect our health over the long run. These activities usually vary from reading, listening to music, watching movies, playing videogames to sleeping. In particular, over the last years, multiple research studies have demonstrated how regular sleep and short naps during the day can improve our well-being and therefore affect our daily activities, above all work, which plays a central role in shaping the mental health of an individual.

On top of this premise, the original idea for this project was to develop an application for smartphones that could help commuters relax during their journey from and back home. The functionality of the program allows the user to select departing and arriving stations, and algorithms automatically keep track of every stop and send a notification when final destination is approaching, with the ultimate goal of relieving the user from the stress of counting stops.

1.2 Available approaches

To make such a function possible, the application could have relied on GPS to collect user's location coordinates and confront them with a database of coordinates for all stations. The same result could be achieved with cellular or Wi-Fi connectivity; however, while travelling on the tube, data signal is usually scarce or absent. For this reason, in order to achieve a similar result, the implemented application uses a combination of real-time data collected from TfL via its Open Data API, and machine learning for sound classification. Eventually, the application downloads information for all stations to allow planning, and once the user has started his journey, the built-in microphone inside the smartphone collects samples of audio and passes it to an algorithm that classifies sound as "Still" or "Moving", permitting the app to track stations along the predefined path. For extended accuracy, the machine learning algorithm is coupled with a timer lasting the exact duration estimated by TfL.

1.3 Roadmap

The remaining part of the report will be structured into five main chapters. The first one will examine background concepts required to understand how an iOS app is developed. These will include design patterns and frameworks. The second chapter will focus on the requirements for the app, while the third section will cover features development. The fourth chapter will show the implementation of the app under the MVC architecture, and the last part will evaluate the program in terms of performance after refactoring, and reliability in real-case scenarios.

2. Structure of an iOS app

Along with Android, iOS has become the predominant mobile operating system. The reason behind this success, is a combination of Apple's commitment to the quality standards of its ecosystem, through annual updates to the OS and the SDK, and a supportive community of creative developers. As of writing, iOS has reached its 13th iteration since its inception in 2007. Over the years, the company has put great efforts in making possible for every developer to implement code for UI or more complex features, such as machine learning or augmented reality, by providing a series of frameworks, packages of code that set a standard interface, and libraries to address these issues. Throughout this project report, some of the frameworks that were used in the development process of the app will be discussed.

2.1 iOS SDK and Swift programming language

In order to write applications for iOS, a developer needs to turn his attention to the iOS Software Development Kit, a suite of tools that form the development environment for creating applications for iPhone, iPad, Mac, Apple Watch and AppleTV. Inside this package, Apple provides an IDE, called Xcode, and an iPhone simulator to run and test prototypes of the app in development before its final release. In particular, Xcode allows managing everything related to the app, from classes that set the functionality and structure of the program, to properties files to access hardware resources, and eventually the GUI through the built-in Interface Builder. In doing so, Apple provides an integrated environment in which features can be implemented either programmatically or through a combination of code and drop-down menus, all of which will be shown in this report.

Ultimately, the standard programming language is the relatively new open-source Swift, a statically typed multi-paradigm language developed by Apple, that inherits features from the previous company's standard language, Objective-C, and other, such as C, making it fast, safe and suitable for the object-oriented development context. Swift supports variables and constants which are always initialised before use through their own reserved words *var* and *let*. Variables cannot be *nil* but an *optional* value, represented by *?*, tells the compiler that the variable could be *nil*, forcing the developer to unwrap the variable to check if it is not *nil* by using *!* if he intends to use it. Additionally, Swift provides support for many other features, such as generics, structs, enums, functional programming, and error handling for extended expressiveness.

2.2 MVC design pattern

By default, when creating a new project on Xcode, the developer is presented with the Model-View-Controller (MVC) design pattern where each object is assigned one of three roles, model, view or controller, and rules to communicate with objects from one of the three layers. This design pattern is critical to the development of an app as it ensures code reusability, extensibility, and loose coupling.

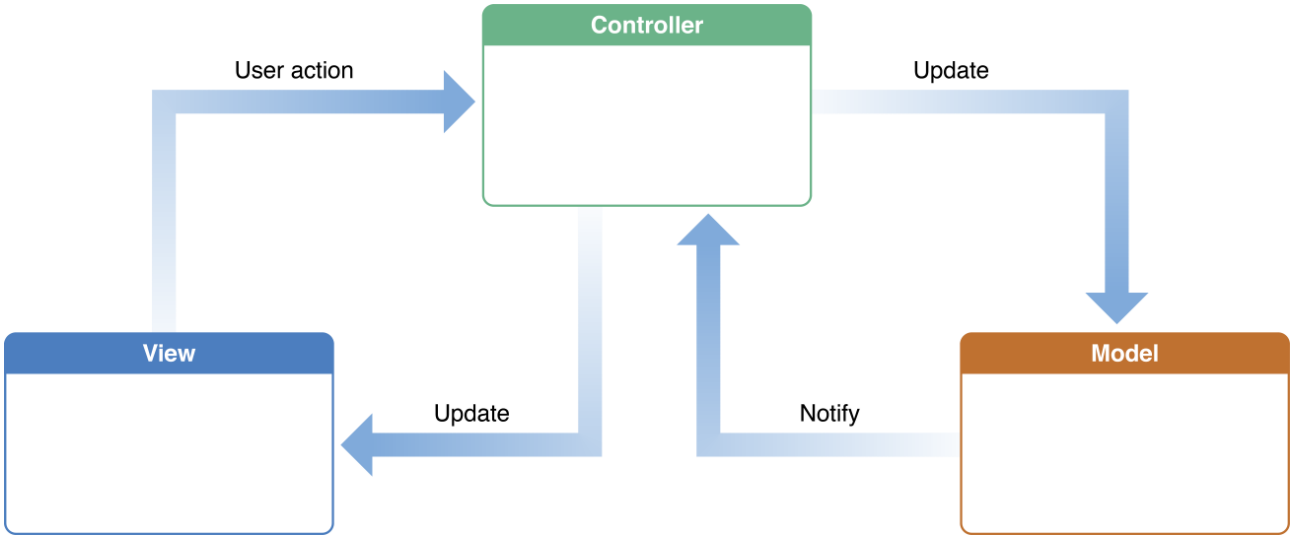

Figure 1 Diagram of MVC model - Source Apple Documentation

When building an app, one has to refer to this model when creating classes. In particular, the Model layer holds the data specific to the app and functions for manipulating it. Data represented here can either be persistent, stored in a database or file, or be transient, only created and destroyed at and after run time. A data representation from the Model layer, can have single or multiple relationships with other structures inside the layer, and because each data model is domain-specific in terms of

knowledge representation, this has to be detached from the user-interface definition, so to be ported and used on other cases that share similarities in terms of data representation. Successively, the View layer contains objects that are a concrete representation, hence graphical elements in the app. These objects enable data from the Model layer to be displayed, and to respond to user actions. Apple provides UIKit framework for *view* classes and Interface Builder for easily instantiating *view* objects, like buttons, labels, fields, tables, search bars or navigation bars. The Controller layer, is responsible to containing objects that act as intermediaries between the Model and View layers, allowing them to communicate changes of their states, for example, user-enabled changes like text entered in a field or new data received over a network. Lastly, the Controller layer manages the life cycle of the objects of the other two layers.

2.1.1 MVVM and MVP instead of MVC

Apple's MVC design pattern has often been criticised for the Controller and the View being tightly coupled, and for the Model being exposed to the View since the ViewController contains the View and updates the Model, all of this making unit testing more difficult. Two additional system design variants derived from the MVC; these are the MVP (Mode-View-Presenter) and the MVVM (Model-View-ViewModel) and revise the previous model by not pivoting around the role of the ViewController. In the case of the MVP, the Model acts as usual, the View is a passive interface for displaying data and routing user inputs, and the Presenter contains the UI business logic and takes data from the model and formats it for the View. The peculiarity of this model is that all three components talk to each other through interfaces, making the parts decoupled. On the other hand, the MVVM uses a binder instead to communicate changes between the ViewModel, interfacing with the Model, and the View. However, for this project, Apple's MVC system architecture was followed by dividing the project folder into three additional folders named Model, View, Controller.

2.2.2 Communication

When following an architectural pattern, it is necessary to have the parts communicate among them, and iOS achieves it through delegation and notification. The former is a design pattern in which a delegate object executes a group of methods on behalf of another object called the delegate by keeping a reference to it. The control object answers the delegate's questions by informing it of impending events that need to be handled, and that can change the state of the delegate itself. The view asks the questions and encodes them with a protocol, in the form of a list of methods. The latter informs the

other objects, which have subscribed to the receive notification, of occurring events. These recipients of the notification, known as observers, actively respond to the event by changing their appearance or status. Notification and delegation are two powerful mechanisms for obtaining coordination in a program. It reduces dependencies between objects thus promoting code reusability.

2.2 UIKit

UIKit is a framework that provides an infrastructure to construct user interfaces, and it is responsible for interactions with the system, and data and resources management. Each app usually has at least one view for displaying information and interact with its user; in order to do so, each view is connected to a class of type *UIViewController* which is responsible for defining the behaviour of that portion of the app's user interface. When creating a view controller, the class inherits from *UIViewController* a property of type *UIView*; this is one of the most essential classes in UIKit and is responsible for displaying *view* objects - graphical elements - on the screen, through which user interaction happens. Inside UIKit, Apple provides several off-the-shelf UI objects for rendering data; these include labels, tables, buttons, search bars, and frames for displaying images. In order to display them, each object needs to be added as a subview of the view's property through the method *addSubview().* Along with this technique, Apple allows the developer to add objects to the view via Interface Builder and let control them by creating IBActions and IBOutlets.

2.2.3 Interface Builder

Interface Builder is Xcode built-in interface editor that allows developers to design user interfaces without the need to write code. The editor provides a *.storyboard* file with a canvas in which to drop *View Controllers* objects and set the flow of the app by chaining views together. Since UIKit inherits from Cocoa and Cocoa Touch[1] that are built using the Model-View-Controller pattern, UI objects are already separated from the business logic of the app, and the connection between these elements and the code is automatically managed by the system by assigning each *View Controller* object its *UIViewController* class. Additionally, Interface Builder provides an auto-layout system for managing proportions among UI objects in the view, with the result of automatically adapting the graphical user interface for use in landscape or portrait, and for displaying it on different screen sizes.

[1] Cocoa and Cocoa Touch were Apple's former APIs for building apps for iOS and OS X before Swift was released in 2013

2.2.2 IBActions and IBOutlets

IBAction and IBOutlet are two fundamental concepts when building a user interface. An IBAction is any function that is associated with a UI element and it is used to handle events like taps and gestures. Such a function uses the target-action design pattern in which an object holds information about the method (action) to perform and the receiver object (target) responsible for executing the method when a specific event occurs. The opposite is an IBOutlet, which is a variable referencing to a UI component, and it is called to manipulate that element and change its state. For example, a developer might implement a *UIButton* and associates a tap IBAction to it that increases by one unit the number contained in a *UILabel* (IBOutlet). The easiest way to implement both of them is by control-dragging from storyboard to the corresponding swift file containing the *UIViewController* class associated with the view.

2.3 Segue

If an app presents multiple views, in order to transition between them and present a new interface, Xcode uses segues. The simplest form of a segue can be created in Interface Builder by doing control-drag from one item in the view we want to transit from, to the endpoint view for the segue. This form of transition is initiated whenever the user interacts with the item associated with the action, for example, a tap on the screen on a *UIButton*. Different options are then available for this action; a Show action pushes the following view on top of the Navigation Controller stack, thus creating a pile of views allowing the user to navigate back and forth from the *rootViewController*. Another example is Present Modally, allowing the developer to simply render a view on top of the current one without creating any hierarchy with the previous view. However, this way of creating segue is not useful for passing data between view controllers, which might be required when the user selects an option from a list, and the data it contains is needed in the following view controller. One way to accomplish this is using a coded segue, where an action associated with a UI element incorporates code for instantiating the destination view controller, and properties of the variable storing the view controller are called where data needs to be passed. In the case below, for example, an IBAction *preferecesPressed()* is associated with a *UIButton*. When the button is tapped on the screen, the function is called and loads a *ViewController* called SettingsFavourite and successively injects *trip* and *dataController* into the controller's respective variables. The last line pushes the controller on top of the navigation controller stack and displays the new view.

```
    @IBAction func preferencesPressed(_ sender: Any) {
        let controller = storyboard?.instantiateViewController(withIdentifier: "SettingsFavuorites") as! SettingsFavourites
        controller.trip = trip
        controller.dataController = dataController
        controller.previousViewController = self
        navigationController?.pushViewController(controller, animated: true)
    }
```

Source: PlanJourney.swift

3. Requirements

The proposed application was conceived to pivot around the main view through which the user can plan its journey, by entering in a text field the names of the departing and arriving stations, and eventually can set a date for a more personalised and precise search. When entering one of the two stations, the user should be able to select from an updated list of all available stations. When specifications of the trip are set, the user should press a button to trigger a request for planning his journey. The result of this event should switch the interface to a new selectable list of suggested trip plans, displaying lines involved in the option, duration, number of stops and direction of the train. Whenever the user indicates a preference by selecting one of the options, the view is updated with a map showing the itinerary and a list with all the stops in the journey. A button should be added to allow the user to begin the journey and let the system track stops. Additionally, the user should always be allowed to go back to the main view to reset journey's preferences or start a new trip. Ultimately, two views should complement the main one and be accessible from it; the first one should display a list of favourites for a quick set up of the journey and allow the user to set parameters for filtering search results. A second one should present a series of images depicting individual lines with their related names for information purposes.

3.1 Data

To obtain specifications for all stations, one could hardcode them in a dictionary or store information in a database and retrieve it whenever the app launches. Another option was to check if this type of data was available through an API. Although the first solution avoids any type of data connectivity, it is extremely inflexible and requires a constant update by the developer if any of the lines or stations were to present issues like delays or planned works. On the other hand, an optimal API should provide real-time data on the status of each line and allow access to a database of all lines and stations. Conversely, this option requires an internet connection which a user is presumed to have before entering any station were signal is usually scarce.

3.2. Journey Plan

In order to obtain route options for the trip, again, one could resort to algorithms or APIs. The first solution would have required to build a data structure capable of imaginarily represent the metro network. For this scope, for example, an adjacency matrix constructed with a two-dimensional array, storing a boolean at each position indicating whether two stations are connected, would be a good solution for representing the graph structure of the network. Each entry of the matrix would indicate a station, hence a node in the graph, while the boolean value would be the equivalent of an edge. Then, in order to get the optimal path from the matrix, a depth-first search algorithm would be performed, where starting from a node, each of its vertex is visited until a connection to another node is found, and the process is repeated until the final node is reached. Unfortunately, this method could not cope with situations where a station is momentarily inaccessible. Alternatively, a specific API for planning a trip would have real-time access to lines' status and therefore return an optimal and valid result.

4. Development

4.1 Persistent storage

To make the experience faster and more personalised, a persistent store was included to allow users to store their favourites journeys, allowing them to easily set options for search once they have opened the application. Usually, a program's memory is always cancelled after runtime, and in order for the user to store information permanently, a program needs to rely on a database from which to retrieve and record data. Xcode proposes two ways for permanently storing information, each being more convenient depending on the task we want to accomplish.

Before continuing, it is important to understand how iOS file management system works. The core principle is sandboxing where each application is treated as a self-contained and independent unit, thus protecting the system from malicious code and limiting app's privileges to its intended functionality. Each sandbox is allocated a separated space in memory with respect to the OS and other applications that the user might have installed. Here space is originally divided into several premade containers; for example, the Bundle Container contains the executable file and the app's resources

like images and audio files. The Data Container holds all the user and app's data by distinguishing among Temporary, data that does not need to be persisted, Documents, user's data, and Library, information hidden from the user.

The first solution uses a *UserDefaults* object. As the name suggests, this is a dictionary of key-value pairs that are extremely useful for storing users' preferences. It can contain data in the form of strings, numbers, dates, arrays or dictionaries but all of this is limited by the fact that data is read and written in a single block, making this option less viable for large files. The second solution requires the developer to import and work with Core Data, Apple's framework for permanent data for offline use and for caching temporary data, representing a great solution when data has structure and relationships. To save information, a persistent store is needed, and iOS supports three types. The default implementation is an SQLite store, then binary store and in-memory store are also available. The former is appropriate when read and write of the database is performed in its entirety, for example, when using .csv files. The latter is useful when data is small and can fit memory all in one, like in the case of caching. Eventually, an important aspect of Core Data is that it abstracts the persistent store by putting a layer in between the database and the user, making it easy to save and fetch data without being concerned about the complexity of the underlying infrastructure.

For the implementation of the persistent store, Core Data provides a set of four classes - the data stack - that work together to manage the data layer. These parts are the Managed Object Context, the Managed Object Model, the Persistent Store Coordinator, and the Persistent Container. The first class is useful for managing objects that we want to add, edit or delete by saving or rolling back changes to the persistent store. Changes made in the context are not automatically saved to disk or do not automatically change the UI. When instantiating an object, it is important to pass the context for communication with the store. Managed Object Model defines entities, their attributes and their relationships with other entities. This part can be hard-coded, but Xcode provides an intuitive data model editor interface for creating and managing specifications of the database. The Persistent Store Coordinator is the agent that connects to the database and uses the model to mediate between the context and the store itself by converting store records into managed objects and vice versa. The last piece of the stack helps in setting up the whole structure by proving useful variables and methods for working with context and reducing boilerplate code.

In the case of TubeNap, an SQLite database for storing favourites was implemented by creating a new *Data Model* file called DataModel. To set up the Core Data stack, a class called *DataController*

was created in the *DataController.swift* file by importing Core Data. The aim of the class is to hold a persistent container instance, help load the persistent store and access it. The constant *persistentContainer* is assigned an object of type *NSPersistentContainer* that is then initialised by passing the name of the data model that was created to store favourites. A *load()* function was created to load the persistent container, along with an additional computed-property variable *viewContex* returning the persistent container's view context. The class was then used in the *AppDelegate.swift* file, responsible for managing special states of the application, like launch, background or termination, in the function *DidFinishLaunchingWithOptions()*.

```
9  import Foundation
10 import CoreData
11
12 class DataController {
13
14     let persistentContainer:NSPersistentContainer
15
16     var viewContext:NSManagedObjectContext {
17         return persistentContainer.viewContext
18     }
19
20     let backgroundContext:NSManagedObjectContext!
21
22     init(modelName:String) {
23         persistentContainer = NSPersistentContainer(name: modelName)
24         backgroundContext = persistentContainer.newBackgroundContext()
25     }
26
27     func configureContexts() {
28         viewContext.automaticallyMergesChangesFromParent = true
29         backgroundContext.automaticallyMergesChangesFromParent = true
30
31         viewContext.mergePolicy = NSMergePolicy.mergeByPropertyStoreTrump
32         backgroundContext.mergePolicy = NSMergePolicy.mergeByPropertyObjectTrump
33     }
34
35     func load(){
36         persistentContainer.loadPersistentStores { storeDescription, error in
37             guard error == nil else {
38                 fatalError(error!.localizedDescription)
39             }
40             self.configureContexts()
41         }
42     }
43 }
```

Source: DataController.swift

The structure of the database is simple and contains a single entity called *Favourite* with two *String* attributes for departure and destination. The table is then referenced in *SettingsFavourites.swift* and *CurrentJourney.swift* files. In *SettingsFavourites.swift, a* special *fetchFavourites()* function was

coded to perform favourites retrieval to populate the *UITableView* in SettingsFavourites Scene in the *Main.storyboard*.

```
40      func fetchFavourites(){
41          let fetchRequest : NSFetchRequest<Favourite> = Favourite.fetchRequest()
42          let sortDescriptor = NSSortDescriptor(key: "departure", ascending: false)
43          fetchRequest.sortDescriptors = [sortDescriptor]
44          if let result = try? dataController!.viewContext.fetch(fetchRequest){
45              favouriteList = result
46              favouriteTableView.reloadData()
47          }
48      }
```

Source: SettingsFavourites.swift

The code below shows the function being called (line 30) after loading the view in the View Controller. The function (cf. above code) declares a variable of type *NSFetchRequest* (line 41) for performing data fetch. It then assigns the object's *sortDescriptors* property an *NSSortDescriptor* object to set the parameters of the closure (lines 42-43). The result is then sorted and assigned to a variable of type *List* called *favouriteList* that contains data from which the Table View retrieves its content (lines 45-46)

```
25      override func viewDidLoad() {
26          super.viewDidLoad()
27          navigationController?.delegate = self
28          timeDepartingArriving.isOn = timeISSwitch()
29          settingsLeast.selectedSegmentIndex = setPreferenceLeast(preference: trip!.journeyPreference)
30          fetchFavourites()
31      }
```

Source: SettingsFavourites.swift

Always in *SettingsFavourites.swift* file, *deleteFavourite()* function shows how simple it is to delete a record from the persistent store. The cell containing the reference to the entity's record is selected, and subsequently, the method *delete()* is invoked on *dataController* of type *DataController* by passing the content of the cell. Again the context is saved to confirm changes and a call to *fetchRequest()* is push onto the stack to *reloadData()* on the *UITableView*.

```
101     func deleteFavourite(at indexPath: IndexPath) {
102         let favouriteToDelete = favouriteList[(indexPath as NSIndexPath).row]
103         dataController!.viewContext.delete(favouriteToDelete)
104         try? dataController!.viewContext.save()
105         fetchFavourites()
106     }
```

Source: SettingsFavourites.swift

The other scenario in which the database is used is when the user writes to it to add to Favourite table the departing and arriving stations for a journey he has previously selected after planning a trip. In this case, a function *add()* simply calls a reference to the entity Favourite in the database (line 269),

assigns to each of its attributes the corresponding station, and successively save the context to confirm the operation.

```
268     @objc func add(sender: UIBarButtonItem) {
269         let favourite = Favourite(context: dataController!.viewContext)
270         favourite.destination = trip!.destination
271         favourite.departure = trip!.departure
272         try? dataController!.viewContext.save()
273         alertForAdditionToFavourites()
274     }
```

Source: CurrentJourney.swift

4.2 TfL data request

Transport for London (TfL), the local government body responsible for the transport system in Greater London, offers access to a stream of meaningful and real-time data through its Open Data API. The structure of the data is homogeneous across all modes of transport, allowing the developer to write code that fits all cases. The API output for a request can either be in JSON or XML formats. Before parsing data, one needs to look at the response output for a particular endpoint of interest. TfL offers several endpoints that spans from requests for air quality, traffic on roads, space available at specific parking areas, to anything related to the bus and tube networks. In the case of TubeNap, data was requested in JSON, as Apple makes it easy to read data in this format by simply instantiating a *JSONDeconder* object from the Foundation framework. In order to decode the output from a request, it is necessary to pass the standardised JSON structure of the output (for demonstration purposes, I will refer to a GET request for all lines' names from the metro network. The same pattern has been applied to the other request functions inside *TflRequest.swift* file and the JSON folder). Therefore, in a separate *AllTubeLines.swift* file (cf. figure 2), a struct representing response output for the GET request (cf. figure 3) was implemented. A *JSON* object is represented as a dictionary of keys and values. Sometimes, the response file can be a list of *JSON* objects, or a key can reference another *JSON* object. In the case of AllTubeLines, the *Line* struct had to conform to the *Codable* type, an alias for the Encodable and Decodable protocols which distinguish between POST or GET requests respectively. Subsequently, the struct lists a series of properties matching the key-value pairs in the response *JSON* object. For example, line 66 of the struct, shows a constant named *type* of type *String,* matching the output of the third line of the JSON response file. Different is the case at line 74 of the struct, where *routeSection* is a *List* of type *RouteSection*, meaning that the output from the request is another *JSON* object that needs to be appropriately decoded with another struct representing the *JSON*

object. An enum follows the struct's properties and it is used to match any constant's name to the name of the keys in the *JSON* object, trough the *CodingKey* type for encoding and decoding.

```swift
struct Line : Codable {

    let type : String?
    let id : String
    let name : String
    let modeName : String
    let disruptions : [String]?
    let created : String
    let modified : String
    let lineStatus : [String]?
    let routeSection : [RouteSection]?
    let serviceTypes : [ServiceTypes]?
    let crowding : Crowding2?


    enum LineKeys : String, CodingKey {

        case type = "$type"
        case id = "id"
        case name = "name"
        case modeName = "modeName"
        case disruptions = "disruptions"
        case created = "created"
        case modifed = "modified"
        case lineStatus = "lineStatus"
        case routeSection = "routeSection"
        case serviceTypes = "serviceTypes"
        case crowding = "crowding"

    }
}
```

Figure 2 TubeNap source code - AllTubeLines.swift file


```json
[
  {
    "$type": "Tfl.Api.Presentation.Entities.Line, Tfl.Api.Presentation.Entities",
    "id": "bakerloo",
    "name": "Bakerloo",
    "modeName": "tube",
    "disruptions": [],
    "created": "2019-09-10T12:56:38.367Z",
    "modified": "2019-09-10T12:56:38.367Z",
    "lineStatuses": [],
    "routeSections": [
      {
        "$type": "Tfl.Api.Presentation.Entities.MatchedRoute, Tfl.Api.Presentation.Entities",
        "name": "Harrow & Wealdstone Underground Station - Elephant & Castle Underground Station",
        "direction": "inbound",
        "originationName": "Harrow & Wealdstone Underground Station",
        "destinationName": "Elephant & Castle Underground Station",
        "originator": "940GZZLUHAW",
        "destination": "940GZZLUEAC",
        "serviceType": "Regular",
        "validTo": "2019-10-06T00:00:00Z",
        "validFrom": "2019-09-07T00:00:00Z"
      },
      {
        "$type": "Tfl.Api.Presentation.Entities.MatchedRoute, Tfl.Api.Presentation.Entities",
        "name": "Elephant & Castle Underground Station - Harrow & Wealdstone Underground Station",
        "direction": "outbound",
        "originationName": "Elephant & Castle Underground Station",
        "destinationName": "Harrow & Wealdstone Underground Station",
        "originator": "940GZZLUEAC",
        "destination": "940GZZLUHAW",
        "serviceType": "Regular",
        "validTo": "2019-10-06T00:00:00Z",
        "validFrom": "2019-09-07T00:00:00Z"
      }
    ],
    "serviceTypes": [
      {
        "$type": "Tfl.Api.Presentation.Entities.LineServiceTypeInfo, Tfl.Api.Presentation.Entities",
        "name": "Regular",
        "uri": "/Line/Route?ids=Bakerloo&serviceTypes=Regular"
      }
    ],
    "crowding": {
      "$type": "Tfl.Api.Presentation.Entities.Crowding, Tfl.Api.Presentation.Entities"
    }
  },
```


Figure 3 Output of a GET request for all lines' names performed on Postman

Below code for fetching lines' names is shown. After instantiating a *JSONDecoder* object, to perform the GET request, we assign two new variables of type struct that matches the JSON response, and of type *Data* that takes the endpoint as an argument. Successively, in a do-catch block, the method *decode()* is called from the *JSONDecoder* object, and the variables of type *Line*[2] and *Data* are passed as arguments. The result is appended to a *List* of *String* and then returned as result of the fruitful function.

[2] Note that object of type Line is contained inside square brackets as the expected result is a list of JSON objects (cf. Figure 3)

```swift
func requestAllLinesName() -> [String] {

    var response = [String]()

    let decoder = JSONDecoder()
    let tflLines : [Line]
    let data: Data = try! Data.init(contentsOf: URL.init(string: "https://api.tfl.gov.uk/Line/Mode/tube/Route")!)

    do {
        tflLines = try decoder.decode([Line].self, from: data)
        for station in tflLines {
            response.append(station.name)
        }
    }catch{
        print(error)
    }
    return response
}

```

Source: TflRequests.swift

4.3 Sound Classification

Apple makes it easy to implement machine learning into iOS, thanks to the Core ML framework. Although the company has some models for speech recognition and image classification, neither of these is suitable for what is required in the app, which is sound classification[3]. Sound classification is the task of assigning labels to detected sound, and in the case of TubeNap the goal is to distinguish between sound that indicates a still or moving carriage while performing a journey on the tube[4]. Relying on this technique to track the train along its predefined path, seems to be a feasible task for reasons I will show. In order to do so, some prior steps were required. Firstly, data collection was needed and therefore audio samples were collected with iPhone's Memo app, either through the built-in microphone or accessories, ensuring that audio quality was the same as when performing sound classification in real-case applications. Records include journeys on all lines, either at peak or off-peak time, with the aim of creating a rich and random dataset. Audio samples of other tube networks from different cities were also included with the intent of making the model as general as possible.

After collection, data exploration was performed by analysing audio spectrograms[5] for each record, to check presence of any distinguishing feature between the samples. The result is presented below to show the difference in frequencies produced by a train on the Central Line, between Bank and St

[3] With the release of iOS 13, Apple has revealed new models, including one for sound classification which at the time of writing is still in beta and therefore I have preferred to implement other third-party models

[4] As mentioned in the proposal for this project, the reason behind implementing such a feature, comes from the need of tracking the train in the case of absence of data connectivity, that being cellular, Wi-Fi or even GPS for geolocation

[5] Spectrogram, or sonograph when applied to audio, is a type of graph used to show the variance in frequency of a data source through time

Paul's stations, over a period of five seconds. The graphs clearly show evidence that the two moments produce different frequencies, making audio a viable source of information to perform tracking.

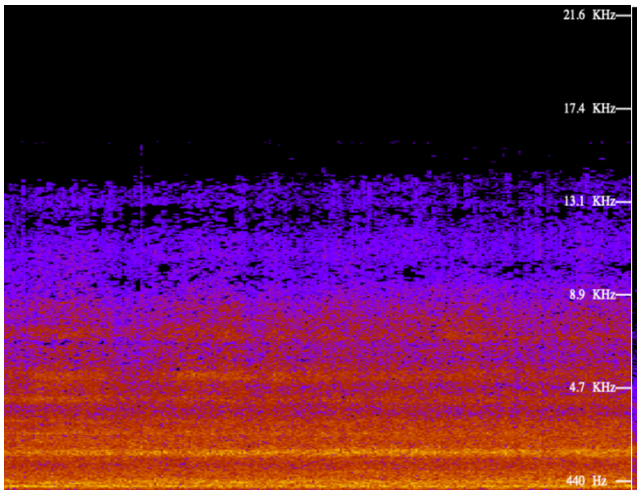

Figure 4 Spectrogram while still at Bank station

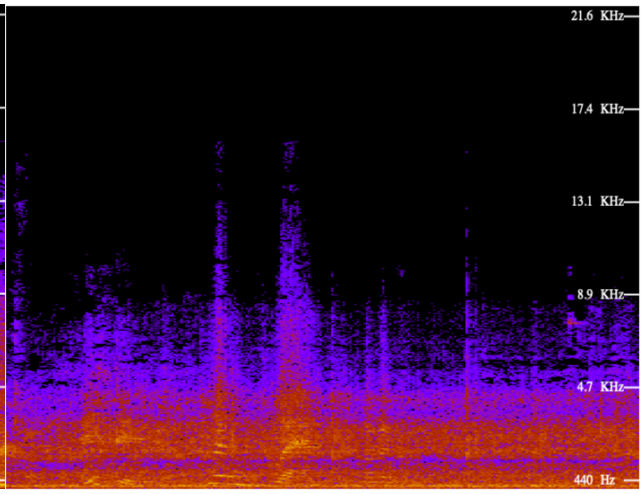

Figure 5 Spectrogram while moving between Bank and St Paul's

After analysing the samples, a third-party machine learning model for sound classification was researched. Multiple sources, like AWS, Google's Tensor Flow or IBM, offered a solution to the task, but all of them required data to be uploaded online for training and prediction, and with privacy and connectivity constraints in mind, a model from Turi Create seemed a better fit. Turi Create is a subsidiary of Apple, specialised in artificial intelligence, that creates and offers easy-to-implement custom machine learning algorithms for common tasks like image classification, object detection, and eventually sound classification. Turi declares its algorithms to be 5-step recipes for performing the specified task, were the actual implementation of the model is abstracted away to let the developer focus on supplying the right data. Before generating the algorithm, data needed to be labelled and cleaned to conform to the model specifications which requires audio to be WAV format, 5-second long, single-channel, 16-bit at 44.1 kHz wave frequency[6]. For this purpose, a folder containing the audio samples, 35 moving and 37 still, and a .csv file with labels referencing the samples were created. Below, a Python algorithm for training the model is shown before exporting it into Xcode:

[6] The task was completed with the help of: https://audio.online-convert.com/convert-to-wav

```
1  import turicreate as tc
2  from os.path import basename
3
4  # Load the audio data and meta data.
5  data = tc.load_audio('direcotory of audio')
6  meta_data = tc.SFrame.read_csv('direcotory of .csv file')
7
8  # Join the audio data and the meta data.
9  data['filename'] = data['path'].apply(lambda p: basename(p))
10 data = data.join(meta_data)
11
12 # Make a train-test split, exclude first fold from test set.
13 test_set = data.filter_by(1, 'fold')
14 train_set = data.filter_by(1, 'fold', exclude=True)
15
16 # Create the model.
17 model = tc.sound_classifier.create(train_set, target='category', feature='audio')
18
19 # Generate an SArray of predictions from the test set.
20 predictions = model.predict(test_set)
```

Source: Turi Create User Guide

The algorithm starts by importing the necessary libraries. In then proceeds by loading the audio samples and their labels from the specified directories. Next, each audio file is associated with its label (line 9), and the dataset is split into test and training sets (lines 13-14), used for generating the model and testing it respectively. The generated model (line 17) is a Neural Network which is a form of machine learning where an algorithm starts by knowing nothing about the data being passed, and over time it gradually learns from input data by setting parameters that help predict new data by improving accuracy. Lastly, the model is then tested (line 20) to check performance with data not used when generating the model. Table 1 shows results from the first round of training, done after splitting the database in half. The numbers show an accuracy of 55%, indicating that the model is good at predicting a moving carriage, while the contrary is not true, and false positive (predicting moving instead of still) tend to be high, suggesting to review samples for the still case and collect new data for improving accuracy.

Data:

target_label	predicted_label	count
still	moving	13
moving	still	2
still	still	5
moving	moving	14

Table 1 Performance of the algorithm after first training

4.3.1 Improvements to the model

To improve the model, an understanding of how sound classification works is needed. The process is usually split into three phases, with the first being signal processing, the second deep features extraction from a pre-trained neural network model, and the last, fitting the custom neural network model for prediction. The first step will be described when presenting the code that has been integrated from Turi Create reference guide. The second step involves using a pre-trained convolutional neural network from Google, called VGG, which has proved to be good at image and sound classification tasks, to extract features that are then passed as the initial parameters of the custom model, thus simplifying the training process. The last step is the only one where actual updates to the model happen, as new data comes in for training. Therefore, to tune the model, 21 new samples were collected for training and testing, and variation to the model were experimented to check for any prediction improvements. The updated dataset, now containing 93 samples, was split into five folds for cross-validation[7], a technique used in statistical analysis to check a model's generalisation ability. Greatly improved results presented in Table 2, allow spotting on which samples the model tends to be weak by identifying the respective fold. By listening to those records, one can notice that the samples might present interferences from other nearby trains approaching, or the sound might be muffled by the person moving and the phone being in the pocket.

Fold 1

accuracy': 0.8333333333333334

target_label	predicted_label	count
still	moving	2
moving	still	1
still	still	7
moving	moving	8

Fold 2

accuracy': 0.8888888888888888

3 layers - 100 nodes each

target_label	predicted_label	count
still	moving	2
still	still	7
moving	moving	9

Fold 3

accuracy': 0.5789473684210527

target_label	predicted_label	count
moving	moving	1
moving	still	8
still	still	10

Fold 4

accuracy': 0.5625

target_label	predicted_label	count
still	moving	4
moving	moving	4
moving	still	3
still	still	5

Fold 5

accuracy': 0.8

target_label	predicted_label	count
moving	still	4
still	still	10
moving	moving	6

Table 2 Cross-validation results

[7] Cross-validation testes model's accuracy by passing new data that was not used is the training process. The data is generally split in k-groups, and iteratively the model is then trained on k-1 groups and tested on the remaining fold. This permit to check whether the model suffers from overfitting or selection bias, and eventually evaluate generalisation when presented with new data.

In addition to cross-validation, the structure of the neural network was tweaked by trying to add layers and neurons to each layer[8]. Results from testing the model on three different structures, by using the first fold as test set, are shown below. The table suggests that adding layers do not affect performance, while increasing density of each layer seem to have a positive impact on accuracy, therefore making the 2-layer and 1000 nodes model, the final choice for TubeNap. It is important to notice that although the model shows a high degree of accuracy, the tracking feature of the app cannot rely entirely on the output of the classification in order to count stops. This because, there can be cases in which the prediction is incorrect, or other cases that do not depend on classification rather on issues relative to the train, for example, delays when the train stops in the middle of the path between two stations. The fifth chapter will show how the model was integrated and how classification results are handled in the tracking process.

2 layers - 100 nodes each - Default settings

accuracy': 0.7777777777777778

target_label	predicted_label	count
still	moving	3
moving	still	1
still	still	6
moving	moving	8

3 layers - 100 nodes each

accuracy': 0.7777777777777778

target_label	predicted_label	count
still	moving	3
moving	still	1
still	still	6
moving	moving	8

2 layers - 1000 nodes each

accuracy': 0.8333333333333334

target_label	predicted_label	count
still	moving	2
moving	still	1
still	still	7
moving	moving	8

Table 3 Results from testing on three different structures

4.3.2 Exporting the model to Xcode

After training, Turi allows to export the model using a built-in method, which creates an *.mlmodel* file that is ready for use in Xcode by simply dragging it into the project main folder. The model was expressively used in *CurrentJourney.swift* file which is responsible for coordinating functions related to displaying and updating the current status of the journey. To start, CoreML and AVFoundation, a

[8] The idea here is that by adding layers and neurons, the algorithm goes more in depth in understanding the features that make up each category so increasing accuracy.

framework for dealing with time-based audiovisual media, were imported; then, a variable *audioHandler* was assigned type *AudioHandler*, which is a struct containing 5 variables that will be responsible for getting the input from the microphone, saving the file in .wav format and pass the sample to the model. The following code snippet shows the algorithm to collect audio and process it for real-time prediction:

```
409     func startRecording() {
410         //1
411         audioHandler?.recordingSession = AVAudioSession.sharedInstance()
412         try! audioHandler?.recordingSession!.setCategory(.record, mode: .default)
413         try! audioHandler?.recordingSession!.setActive(true)
414 
415         // 2
416         audioHandler!.audioURL = CurrentJourney.getWhistleURL()
417 
418         // 3
419         let settings = [
420             AVAudioFileTypeKey: Int(kAudioFileWAVEType),
421             AVSampleRateKey: 16000,
422             AVNumberOfChannelsKey: 1,
423             AVEncoderAudioQualityKey: AVAudioQuality.high.rawValue
424         ]
425         // 4
426         audioHandler?.recorder = try! AVAudioRecorder(url: (audioHandler!.audioURL!), settings: settings)
427         audioHandler?.recorder!.delegate = self
428         audioHandler?.recorder!.record()
429     }
```

Source: CurrentJourney.swift

The first step is completed via *startRecording(),* a no-parameter function responsible for instantiating an *AVAudioSession* object which interacts with the operating system in managing access to audio hardware. *AVAudioSession* is a singleton and it is set under the record category for recording audio. The second step in the function assigns *audioURL* the destination and file's name of the future audio sample. In step three, a constant named *setting* is assigned a list of parameters to tune the audio to the preferred settings required by the machine learning algorithm to work[9]. The last step calls the variable *recorder* of type *AVAudioRecorder* responsible for the actual recording; a URL reference for storing the record is passed, and once it is set, it starts recording (line 426). Finally, method *finish()* is called on *recorder* and to stop recording.

[9] As mentioned before, the model requires audio data to be in WAV format, 16-bit, single-channel,

```swift
func performClassifcation(path: URL){
    do {
        let fileUrl = path
        audioHandler?.wav_file = try AVAudioFile(forReading:fileUrl)
    } catch {
        fatalError("Could not open wav file.")
    }

    print("wav file length: \(audioHandler!.wav_file.length)")
    assert(audioHandler?.wav_file.fileFormat.sampleRate==16000.0, "Sample rate is not right!")

    let buffer = AVAudioPCMBuffer(pcmFormat: audioHandler!.wav_file.processingFormat,
                                  frameCapacity: UInt32(audioHandler!.wav_file.length))
    do {
        try audioHandler?.wav_file.read(into:buffer!)
    } catch{
        fatalError("Error reading buffer.")
    }
    guard let bufferData = try buffer?.floatChannelData else {
        fatalError("Can not get a float handle to buffer")
    }

    // Chunk data and set to CoreML model
    let windowSize = 15600
    guard let audioData = try? MLMultiArray(shape:[windowSize as NSNumber],
                                            dataType:MLMultiArrayDataType.float32)
        else {
            fatalError("Can not create MLMultiArray")
    }

    // Ignore any partial window at the end.
    var results = [Dictionary<String, Double>]()
    let windowNumber = (audioHandler?.wav_file.length)! / Int64(windowSize)
    for windowIndex in 0..<Int(windowNumber) {
        let offset = windowIndex * windowSize
        for i in 0...windowSize {
            audioData[i] = NSNumber.init(value: bufferData[0][offset + i])
        }
        let modelInput = tubeSoundClassifierInput(audio: audioData)

        guard let modelOutput = try? model.prediction(input: modelInput) else {
            fatalError("Error calling predict")
        }
        results.append(modelOutput.categoryProbability)
    }

    // Average model results from each chunk
    var prob_sums = Dictionary<String, Double>()
    for r in results {
        for (label, prob) in r {
            prob_sums[label] = (prob_sums[label] ?? 0) + prob
        }
    }

    var max_sum = 0.0
    var max_sum_label = ""
    for (label, sum) in prob_sums {
        if sum > max_sum {
            max_sum = sum
            max_sum_label = label
        }
    }

    let most_probable_label = max_sum_label
    let probability = max_sum / Double(results.count)
    print("\(most_probable_label) predicted, with probability: \(probability)")
    prediction = most_probable_label

}
```

Source: Three stages of sound classification - CurrentJourney.swift – Imported from Turi Create User Guide

The second part of the prediction task deals with audio processing and sound classification[10]. The function takes the path of the recorded audio sample as an argument, it then opens the file for reading and writing by instantiating an *AVAudioFile* object (line 4). Next, it checks the sample bitrate to be 16 and if positive, a constant *buffer* is assigned an *AVAudioPCMBuffer* for manipulating the record (line 12). Between line 24 and line 44, data from the sample represented in the form of a list of integers, is broken up into several overlapping windows over which the model is applied. The output prediction for each window is a tuple containing a label and a probability. All of them are then sequentially sorted into a dictionary with keys equal to the label of the classifier (line 48-51). From line 55 to line 67, the code checks for the label with the highest frequency of prediction occurrences (line 58) and then simply averages it to present the probability (line 65).

5. Model-View-Controller Implementation

5.1 Model implementation

The model layer contains three classes and one struct. The *DataController* class in the *DataController.swift* file, contains properties and methods for creating the context and loading the persistent container. A background context in the case of multitasking was added to allow the app to continue running while in background. The trip class is responsible for storing crucial information about journey's preferences. Both of these classes are instantiated when the app launches, and successively injected into *PlanJourney* View Controller and then passed between one view and its successive. *TravelOptions.swift* specifies a struct that is used as return value in *PlanJourney()* function. The struct is accessed to obtain information when filling up the cells in *TravenPlan* view, and when displaying map content and stops in *CurrentJourney* view. Lastly, a class called *AudioHandler* was created to handle the creation of audio sessions, recording and saving of the audio sample while running *startClassification()* function.

5.2 Controller implementation

In TubeNap there are six classes that inherit from *UIViewController*, each controlling the logic of one of the views in the user interface. PlanJourney.swift contains the class responsible for setting up parameters and performing the search. In particular, before loading the view, the class initialises two

[10] Given the complexity of audio processing, the algorithm was directly imported from Turi Create's user guide

variables, *trip* and *dataController*, to hold information about the trip, the persistent container and the context it is associated with. After loading the view, delegates protocols of the two text fields are assigned for communication, and their text properties are set equal to departure and destination properties of the *trip* object. If this these are blank, like at launch or when journey's settings are reset, the text fields show a message inviting the user to enter a station. Successively, the *datePicker* is assigned a specific display mode – in the case of TubeNap, date and time – and minimum and maximum bounds are set. Lastly, *date* and *departureHour* properties from the Trip object are passed return values from two custom-made setters, and *canRequet()* function is called to check if planning can be performed. Between line 44 and 78, three functions to display error messages were coded is the case the user inputs incorrect references. Line 101 implements one of the optional methods from the UITextField delegate protocol responsible for communicating that the user has tapped the field and is about to start typing. The function switches interface and displays *TflStops* view from which the user can pick a station. The remaining functions (lines 114-191) are all IBActions associated with the other UI elements, the most important being *Plan Journey* button. When pressed, the function disables all buttons and the activity indicator starts spinning; It then instantiates *TravelPlans* view, calls the function *requestTravelPlanner()* and injects into the corresponding variables of the following view the two variables of type *Trip* and *DataController*. Eventually, the new view is presented by pushing it onto the stack with the NavigationController's method *pushViewController()*. It is important to highlight that the planner request is processed on the global thread, which is a secondary thread on which the developer can run tasks that can be processed in the background in order to avoid the whole UI to be unresponsive. A call to the main thread via *DispatchQueue.main.async* is made to return the result to the main thread[11]. This allows any UI change to happen, like in the case of the activity indicator that stops animating to indicate that the search has produced results and the following view, TflStops, is ready to load.

TflStops class implements the controller layer for *TflStops* view. The class has two variables, *trip* and *dataController*, whose values are injected before presenting the view. In addition, a variable choice is used to temporarily hold the selection from the list, and a variable *senderIsDeparture* is used to signal whether method *didBeginEditing()*, form the previous view, was called by either the departure field or the destination field; In addition, allStopsSearch variable is assigned a copy of the downloaded stops contained in *allStops* property of the *Trip* object, and a constant is used to manage device's location services. Again, this class implements several delegate protocols for using the table view, the search bar and to enable tracking the user's current location. In *viewDidLoad()*, *searchBar*

[11] This is a necessary step as iOS do not allow any UI change to happen outside the main thread

calls method *becomeFirstResponder(),* to immediately show the keyboard when the view in rendered. *tflTable* implements three methods from the delegate protocol for table views, the first one is responsible for indicating the number of rows, in this case equal to the number of elements in the list; *cellForRowAt()* function is used to display content in each row and uses a method – *dequeueReusableCell()* - in which a cell is taken from a queue of available cells that are recycled every time a row disappears from the screen in order to avoid filling the memory with unnecessary data. *didSelectRowAt()* function returns the content of the row and based on the value of the *Boolean senderIsDeparture*, the value is assigned to either departure or destination in *trip*'s properties; Eventually, *trip* and *dataController* are then back injected into the controller, and the view is displayed. *textDidChange()* function from the *searchBar* delegate protocol, takes *allStopsSearch* and applies a filter each time text is typed and reload data with names corresponding to the filter. If instead of choosing a station from the table view, the user presses the current location button, the IBAction (line 132) associated with it, calls *checkLocationServices()* which sets up the location manager by assign its delegate protocol and the measures for estimating current location (line74); it then checks whether location services are enabled for the app and if so, coordinates are passed to their corresponding variables in *trip*. *didChangeAuthorization()* is one of *CLLocationManager* methods responsible for communicating changes to location preferences in the Settings app of iOS.

In *SettingsFavourites* Class, a variable *favouriteList* is used to populate a list with values from the fetch request from Favourite table in the data model. A special *previousViewController* variable is used to keep a reference to the previous *PlanJourney* view controller. When the view is loaded, the fetch request is performed and two helper methods, *timeIsSwitch()* and *setPreferenceLeast()*, are called to determine the status of the switch and the selection of the *UISegmentControl*. Between lines 111 and 131, two IBActions allow switching preferences in the switch and the segmented bar. The table view implements *didSelectRow()* method for assigning departing and arriving stations to *trip* and pass it to the controller of type *Plan Journey*, which is then presented.

ViewNetworkMap uses a *UIScrollViewDelegate* to allow zoom on the images, and minimum and maximum scales of it are set when the view has loaded. An *allLinesMaps* variable temporarily holds reference to a list that is assigned values from *trip*'s *allLines* property. An additional string "Metro Network" is added to the list. An extension to the class, implements methods from the UIPickerView delegate and the data source protocol, both acting similarly to the UITableView. In particular, *didSelectRow()* method assigns *imageView* variable's *image* property a *UIImage* whose name in the asset folder matches the row from the picker.

TravelPlan, initialises two variables *travelPlan* and *directionForOption*, the former is assigned the return value from *PlanJourney()* function, the latter is used to store directions for each option after using *extractDirection()* function in viewDidLoad (Line 31). At line 28 the back button from the navigation bar is hidden and replaced with a cancel button which is assigned object function *cancel()* for returning to the main view. Again, an extension is added to the class to implement methods' bodies for the table view. *cellForRowAt()* uses the custom *OptionJourneyCell* class for rendering data in the cell, and a helper method *returnLineColor()* returns the matching colour for a line's name.

The last class inheriting from *UIViewController* is *CurrentJourney*, which is used to process and render data for *CurrentJourney* view. The class initialises several variables; *JourneyProgress* is used to keep track of journey's progress, *durationList* stores duration of journey's legs, *linesInOption* and *stopsList* contain names of the lines from the previously selected option and all stops for the trip respectively. *stationStringCoordinates* is a list of coordinates tuples expressed as strings. *stationsCoordinatesInPath* is the equivalent of the previous list, but data is represented as *CLLocationCoordinate2D*. Lastly, *prediction* is used to store the result from the machine learning classification algorithm, and *backgroundTask* is a variable of type *UIBackgroundTaskIdentifier* used to notify the app when it has entered background mode. Successively, in viewDidLoad(), *AudioHandler* is instantiated, and the view controller subscribes to *NotificationCenter* (line 59) to know when the app enters background mode. Successively, at line 71 a helper method sets the centre of the map at midpoint among all destinations; between line 73 and74 pins are instantiated from the custom class and placed on the map, and at line 79 the polyline drawing process is done in the global thread. At line 91 the view unsubscribes from the *NotficationCenter* when the view disappears. Between lines 95 and 144, a series of helper methods permit to centre the map when the view is loaded or whenever the user presses a station from the table view (lines 95-122); *addMapTrackingButton()* allows to place a button on the right lower corner of the map and it is assigned a target action (lines 136-144); other functions are used to place a pin or to draw a line on the map (lines 151-171). From line 199 and 246, a series of custom alerts have been created for notifying the user. Ultimately, the IBAction associated with Start button is the main feature of the controller (line 375). Once pressed, a function *timerTrigger*() is called. *startButton* is disabled, the sound classification task is sent to the global thread and the process of recording audio samples is repeated every second over the entire duration of the journey (line 346-350); a timer is then activated and every second, through the object function *progressBarUpdate(),* updates the progress bar at the bottom of the view and checks its status against a threshold to know when it should notify the user of

the upcoming end of the trip (lines 334-339). At the same time, if the journey presents more than one leg, the system starts keeping track of the current leg in the background and notifies the user of any required line change, causing the timer and the classification task to pause (lines 355-368).

```
343     func timerTrigger(){
344         startButton.isEnabled = false
345         registerBackgroundTask()
346         DispatchQueue.global().async { [weak self] in
347             for _ in 0..<Int(self!.timeLeft!)/Int(1.1){
348                 self!.startClassification()
349                 if !self!.timer!.isValid{
350                     break
351                 }
352             }
353         }
354         timer = Timer.scheduledTimer(timeInterval: 1, target: self, selector: #selector(progressBarUpdate), userInfo: nil, repeats: true)
355         if linesInOption!.count > 1{
356             DispatchQueue.global().async {
357                 sleep(UInt32(self.durationsList![0]*60*0.90))
358                 for _ in 0..<Int(self.durationsList![0]*60*0.1){
359                     sleep(UInt32(1))
360                     self.soundNotificaton()
361                 }
362                     DispatchQueue.main.async {
363                         self.alertChangeLine(nextLine : self.linesInOption![1])
364                         self.timer?.invalidate()
365                         self.linesInOption?.remove(at: 0)
366                     }
367                 }
368                 return
369             }
370         if backgroundTask != .invalid {
371             endBackgroundTask()
372         }
373     }
374
```

Source: CurrentJourney.swift

The object function, *progressBarUpdate()*, is also responsible for making sense of predictions from the classification algorithm to track the train. As mentioned in the chapter for implementing the ML algorithm, in order to deal with the uncertainty of the model, when distinguishing between moving (line 304) and still (line 315) in *pogressUpdate() function*'s body, the algorithm keeps track of the sequence of last predictions. In the case of moving, a variable *lastSequenceOfMoving* is used to cancel effects of a still prediction in the case where the model produces a false negative – predicts still. Another variable, *sequenceOfMoving*, is used to signal if the train can move. This is a critical variable when determining if the train has initiated or has resumed its journey. 15, amounting to roughly 15 seconds of travel time, is used as a threshold for evaluation. This last variable is contrasted by an opposite *sequenceOfStill* variable that erases *sequenceOfMoving* in the case where it reaches 7 non-consecutive repetition, meaning that the train has probably stopped at a station. *sequenceOfStill* is always checked in conjunction with *canMove*, which signals if the train can actually move. If *sequenceOfStill* reaches 7, both this variable and *sequenceOfMoving* are assigned value 0, the *Boolean* variable *canMove* is set to *false*, and the stop on top of the table view is deleted and data reloaded from the updated list.

```swift
    @objc func progressBarUpdate(timer: Timer){
        journeyProgress = progressBar.progress + (1/Float(totalDurationInSec!))
        timeLeft! -= 1
        progressBar.setProgress(journeyProgress, animated: true)
        progressField.text = String(Int(journeyProgress*100))+"%"
        if self.prediction == "moving"{
            self.stillCarriage.isHidden = true
            self.movingCarriage.isHidden = false
            sequenceOfMoving += 1
            lastSequenceOfMoving += 1
            if lastSequenceOfMoving > 5{
                sequenceOfStill = 0
            }
            if sequenceOfMoving > 15{
                canMove = true
            }
        }else if self.prediction == "still"{
            self.stillCarriage.isHidden = false
            self.movingCarriage.isHidden = true
            if sequenceOfStill < 7 && canMove == true{
                sequenceOfStill += 1
                lastSequenceOfMoving = 0
            }else if sequenceOfStill == 7 && canMove == true{
                sequenceOfMoving = 0
                sequenceOfStill = 0
                canMove = false
                if nextStop <= stopsList!.count-1{
                  stopsList?.remove(at: 0)
                  stopsTableView.reloadData()
                  if nextStop == stopsList!.count{
                    alertEndOfPath()
                  }
                }
            }
        }
        if progressBar.progress > 0.90{
            soundNotificaton()
        }
        if journeyProgress >= 1{
            timer.invalidate()
            journeyTitle.text = "Arrived at \(trip!.destination)"
        }
    }
```

Source: CurrentJourney.swift

5.3 View implementation

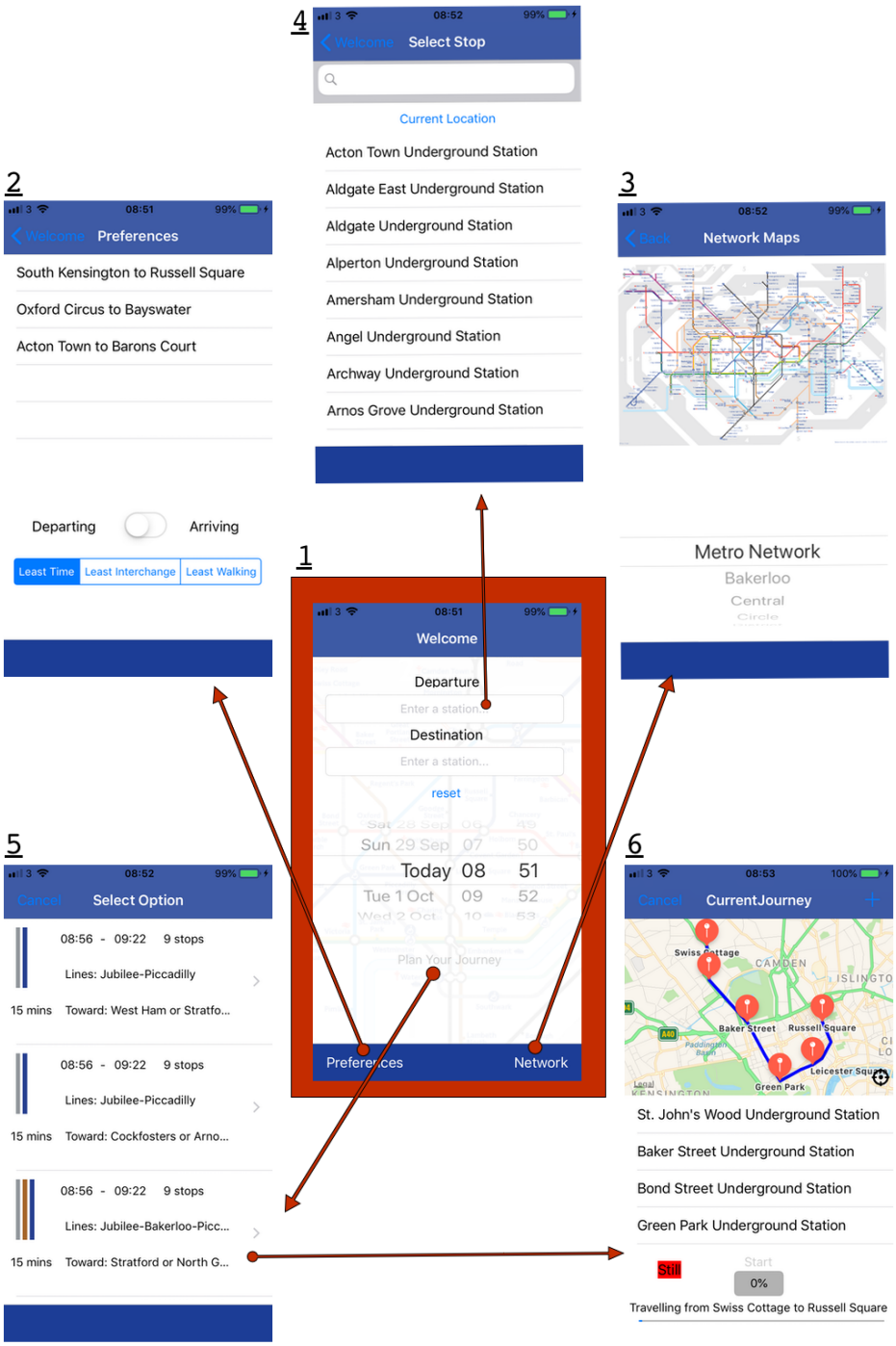

Figure 6 User Interface

An intuitive graphical user interface was conceived in order for the user to interact and visually see results from the system by following guidelines presented in the initial paragraph of the requirements chapter. The interface was composed using Inteface Builder, and it is embedded in a Navigation

Controller. This class inherits from *UIViewController* and it is used to stack a group of views into a hierarchical scheme for navigation. The UI is then is composed of six view controllers, and each instantiates objects from Apple's UIKit framework to graphically display data. When opening the app, a launch screen is presented with a background image of London's tube network. Successively, the main view (1) is presented and the user is prompted to enter the stations by using two *UILabels* and two *UITextFields*. Whenever a field is tapped, the interface switches to a view (4) containing a *UITableView* for displaying elements from a list in a row-like setting, a *UISearchBar* for filtering the list, and a *UIButton* for getting device's current location. On the top left corner, a back button of *UIButton* type allows returning to the initial view. Below the textfields, a *UIButton* is associated with a reset function to clear the fields, and a *UIDatePicker* permits the user to select the date and hour of the trip, where settings have a minimum bound equal to the current date and a maximum of 1 year. Additionally, a *UIButton*, called Plan Journey, if pressed sends a GET request for planning the journey. Finally, at the bottom, the main view presents a *UIToolbar* with two incorporated *UIButtons* for presenting preferences and the network's maps views. The preferences view (2) was composed by placing a *UITableView* on the upper half of the view for presenting favourites and in the bottom half a *UISwitch* and a *UISegmentedControl* for setting parameters for journey search. Oppositely, the network view (3) uses a *UIImageVew* embedded in a *UIScrollView* to present images whose name matches the selected row from a *UIPicker* placed below. The Plan Journey button is enabled only when both stations have been selected. Once it is pressed, the button is again disabled and a *UIActivityIndicatorView* start spinning, indicating that the request was sent, and the system is elaborating the results. The following view (5) presents results in a *UITableView* with custom cells that are defined by the *OptionJouneyCell* class. The *OptionJouneyCell.swift* file contains specifications for the class, which holds six *UITextField* objects for displaying properties from *TravelOption* struct such as the name of the lines in the option, number of stops, duration, departure and arrival stations, and direction. Three *UITextView* objects complete the left side of the cell by matching their background colours to those of the lines in the option. The user can then press a Cancel *UIButton* on the left corner of the screen to go back to the main view and start over the process for a new request or he can continue by selecting one of the rows in the table and move to the last view (6). This *UIView* is split into three parts; the first one contains a *MKMapView* object for displaying the map, the second section has *UITableView* for listing stops, and the third part has one *UIButton* for starting the journey, one *UIProgressBar* to indicate the state of the journey, two *UILabels*, one with a red background and the other with a green one, are used to indicate whether the train is still or moving, and appear only when the result from the prediction matches one of their names. At the top the of the navigation bar, cancel and add buttons respectively allow to stop the journey and return to

main view, and to add the trip's specifications to Favourites table in the database. An additional class form UIKit is used to alert the user in different situations. When the user presses the cancel button during an on-going trip, a pop-up alter is presented on top of the current view with an option to confirm cancellation of the trip or to dismiss the cancel option and remain in *CurrentTrip* view. An alert window, showing a confirmation message, is briefly shown right after pressing the plus button. Lastly, whenever the trip reaches 90% completion, a visual and sound alert, notifying the user of the upcoming end of the trip, is presented with an option to stop tracking the journey immediately. Lastly, *CustomPin* class was added to render pins on the map with the name of each stop from the journey.

6. Evaluation

The app was evaluated under two fundamental metrics, one being usability, meaning how responsive the user interface is under normal use conditions, and the other accuracy, which is if the app does its job of correctly alerting the user. At each new iteration, the app has made big improvements in terms of reactivity. For example, the code below shows how initially, every time the app accessed TflStops view, when loading, the view had to download all lines' names and successively all stations in order to populate the table view, making the interface unresponsive e slow at switching context. In its final version, all the data is downloaded only once at launch time and stored in a Trip variable. The table view is immediately available when switching to TflStops view, making the user feel as if the data were stored internally.

```
29      override func viewDidLoad() {
30          super.viewDidLoad()
31          AllLines.append(contentsOf: requestAllLinesName())
32          for line in AllLines{
33              AllStops.append(contentsOf: requestNameAllStationForALine(lineName:
                    line.replacingOccurrences(of: " ", with:
                    "").replacingOccurrences(of: "&", with: "-")))
34          }
35          AllStops = Array(Set(AllStops))
36          AllStopsSearch = AllStops.sorted()
37      }
```

Source: Early version of TubeNap – TflStops.swift

The change was also influenced by the idea that network coverage is not the same everywhere and data packages with carriers are not unlimited, so making a single download is better than doing at least two, even though the size of the total transferred packages doesn't surpass 0.3Mb in the last

version of the app. Figure 6, shows network requests during a demo; when the app is launched, *trip* is initialised, and data is requested from TfL, with purple bars showing downloads and orange segments indicating GET request. One last data transfer – the isolated bar in the chart - is usually performed when pressing Plan Journey in order to get results from TfL.

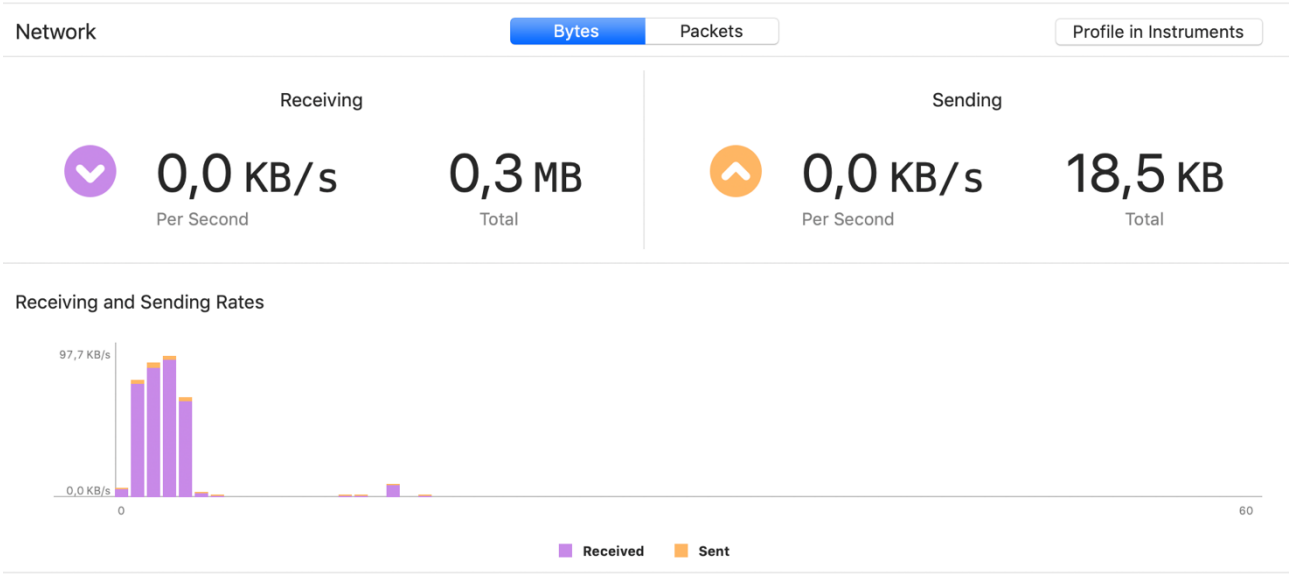

Figure 6 Metrics from Xcode showing network usage over time

Another important leap forward in performance speed was achieved thanks to the use of a dictionary to store key-value pairs of stations and their coordinates in the *Trip* class. This allows for extremely fast retrieval, and benefits are visible in the rendering of the pins and the polyline in the last view. In the beginning, the controller would have made use of a custom function for requesting coordinates for each a station, while now the request is handled at launch and coordinates are stored in a dictionary, and when the user selects a travel option, the list of stops is used to get coordinates. As mentioned before, optimisation was also obtained thanks to multithreading. For example, when the user selects an option, the new view is immediately available, and the polyline feature is pushed onto the global thread and returned only when the task is completed; in the meantime, the user can start the trip because the main thread is available for work. A similar result is achieved with the prediction algorithm, which runs in the background thanks to multithreading. *startClassification()* function in *CurrentJourney.swift* actually sleeps for one second to allow the session to record samples for the ML algorithm; if this were on the main thread, it would have frozen the entire UI and business logic of the controller. Alternatively, when looking at usability, the app was also benchmarked against Google Maps in a case where the user was to launch the app and quickly get information for a trip.

Although Google Maps is richer in terms of feature, so it can be slower at launching, TubeNap seems to be fast enough when compared to Google's service. On a strong 4G connection, Google maps is usually available for use after 1.5-2 seconds, while TubeNap takes 3-4 seconds to launch. This has to do with the fact the TubeNap initialises a *Trip* object which downloads data needed in the app, while Google Maps download data only once the view is loaded and we start typing in the search bar. Additionally, when comparing performance in displaying trip options and the journey window, they seem to perform equivalently. Lastly, in terms of memory usage, the app is not really efficient. Right before selecting an option the app occupies 70MB[12], while after rendering CurrentJoruney view, the number jumps to approximately 140MB primarily because of the graphics involved in the map. Given the nature of the service and the minimal features it implements when compared to Google Maps, the app needs to be optimised for use in lower-end smartphones which might not still have gigabytes of memory available on a normal use case where multiple apps are running in the background. For this reason, when switching from *CurrentJourney* view back to the main view, in *viewDidDisappeared()* function, the map is removed from its view, therefore freeing up space in main memory (line 87).

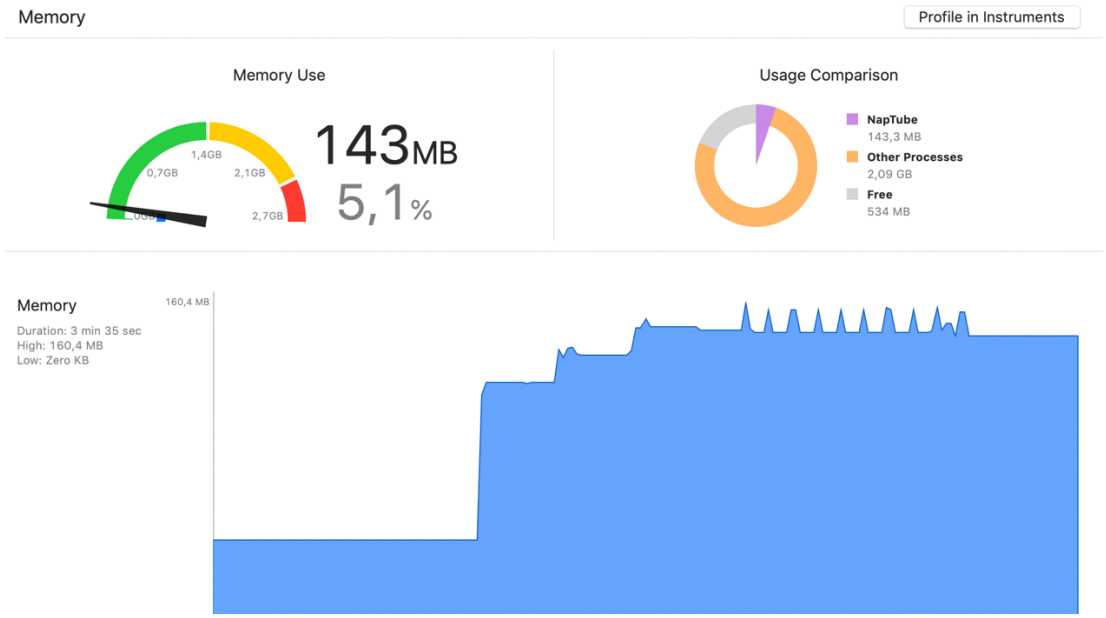

Figure 7 Metrics from Xcode showing memory usage over time

On the other hand, when speaking of accuracy, we have to remember that the goal of the app is to track the train in a case where the device lacks GPS or data connectivity for obtaining its current location, making the task of counting stops reliant on sound classification and simple time count. The app manages to notify the user of the approaching final destination but achieves it thanks to duration

[12] Data refers to demo on an iPhone X

data obtained when downloading real-time information from TfL's API. A 90% threshold for when to start notifying was set for two reasons, one is to give the user the necessary time to become aware of the ending trip, and the second being the fact that actual travel time can be shorter than estimated as some real-case tests have shown. Sound classification seems to work well in combination with condition statements shown before. It is not yet possible to rely uniquely on this approach to track the train, but a combination of both does offer a good experience, making the app fun to use and an interesting investigation of machine learning.

7. Conclusion

The report has described the steps required for developing TubeNap over a period of approximately 12 weeks. The goal of this project was to develop a mobile application for iOS that could help commuters relax during their daily trips to work. The development of the system required to familiarise with several technologies like APIs, frameworks, databases and eventually machine learning. Working under an MVC architecture has shown the benefits of good software development practices, in particular, loose coupling. The overall result is a fully functional system that partially achieves its predefined goal of tracking the train thanks to sound classification. The feature is coupled with a timer to prevent any error that could negatively impact the reliability of the prediction, by making sure that the user is always notified when estimated time of arrival is approaching. The project has also shown how machine learning for classification tasks cannot be used as a black box but rather needs to be adapted for the required objective and be approached as a collaborative effort. Eventually, the prototype app has set the foundation for a more complex system that can be integrated with accessories like the Apple Watch or can be extend with additional means of transpiration, cities and also search results thanks to Google Maps' APIs.

8. References

Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/

Swift Documentation. Ihttps://docs.swift.org/swift-book/. Retrieved from https://docs.swift.org/swift-book/

Model-View-Controller. Retrieved from https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

MVC Vs. MVP Vs. MVVM. Medium . Retrieved from https://medium.com/ios-expert-series-or-interview-series/mvc-vs-mvp-vs-mvvm-13cc6ab43a4c

MVVM Design Pattern - IOS Design Patterns - Raywenderlich.com. YouTube. Retrieved from https://www.youtube.com/watch?v=bFoLlwuzAtk

Notification. Retrieved from https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Notification.html#//apple_ref/doc/uid/TP40008195-CH35-SW1

Delegation. Retrieved from https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

Target-Action. Retrieved from https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/TargetAction.html

UIKit Fundamentals | Udacity. Retrieved from https://www.udacity.com/course/uikit-fundamentals--ud788Swift

UIKit | Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/uikit#//apple_ref/doc/uid/TP40007072-CH4-SW3

IOS Persistence And Core Data | Udacity. Retrieved from https://www.udacity.com/course/ios-persistence-and-core-data--ud325

UserDefaults - Foundation | Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/foundation/userdefaults

Core Data Stack | Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/coredata/core_data_stack

IOS Networking With Swift | Udacity. Retrieved from https://www.udacity.com/course/ios-networking-with-swift--ud421

Unified API. Transport For London . Retrieved from https://tfl.gov.uk/info-for/open-data-users/unified-api?intcmp=29422

TfL Unified API Part 1: Introduction. Digital Blog . Retrieved from http://blog.tfl.gov.uk/2015/10/01/tfl-unified-api-part-1-introduction/

Representing Graph Data Structures. Retrieved from http://archive.oreilly.com/oreillyschool/courses/data-structuresalgorithms/graphDataStructures.html

AVAudioRecorder - AVFoundation | Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/avfoundation/avaudiorecorder

Core ML | Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/coreml

SoundAnalysis | Apple Developer Documentation. Retrieved from https://developer.apple.com/documentation/soundanalysis

Cross-validation (statistics). Wikipedia . Retrieved from https://en.wikipedia.org/wiki/Cross-validation_%28statistics%29#cite_note-6

Turi Create User Guide · GitBook. Retrieved from https://apple.github.io/turicreate/docs/userguide/

Human Interface Guidelines - Design - Apple Developer. Retrieved from https://developer.apple.com/design/human-interface-guidelines/

Multi-Threading Using GCD For Beginners. Retrieved from https://hackernoon.com/swift-multi-threading-using-gcd-for-beginners-2581b7aa21cb

Background Modes Tutorial: Getting Started. Raywenderlich.com . Retrieved September 30, 2019, from https://www.raywenderlich.com/5817-background-modes-tutorial-getting-started