

École polytechnique de Louvain

Coding with the voice

An analysis of the performance of open-source
voice recognition engines

Author: **Damien GONZE**
Supervisor: **Olivier BONAVENTURE**
Readers: **Charles PÊCHEUR, Anthony GEGO**
Academic year 2019–2020
Master [120] in Civil Engineering

Contents

1	Introduction	3
2	Voice recognition for the browsers	5
2.1	Web Speech API	5
2.1.1	Google Chrome	5
2.1.2	Firefox	6
2.1.3	Other browsers	6
2.2	PocketSphinx	7
2.2.1	PocketSphinx in a browser	7
2.3	Google Cloud Speech, Amazon Transcribe, Wit.Ai, etc.	8
2.4	External server	8
2.4.1	Dragon	8
2.4.2	Kaldi	9
2.5	Which voice recognition engine to use	9
3	Algorithms for voice recognition	10
3.1	Voice recognition process	10
3.2	The Viterbi algorithm	11
3.3	Beam search	14
3.4	A*	14
3.5	GMM	16
3.6	SGMM	16
3.7	Phonetic fast match	17
3.8	Other algorithms and more	17
4	Voice recognition for code	18
4.1	Voice recognizer are built for sentences	18
4.2	Code have multiple context	19
4.3	Existing projects	19
4.4	Conclusion	20

5	PocketSphinx	21
5.1	Grammar	21
5.1.1	JSGF	21
5.1.2	FSG	23
5.2	CMU pronouncing dictionary	24
5.3	Web worker and Wasm	24
5.4	Acoustic model	25
5.5	Command line arguments	25
6	Our implementation	26
6.1	A modular library	26
6.1.1	The recognition	26
6.1.2	The modules	27
6.2	Web speech API	29
6.3	PocketSphinx.js	30
6.4	The implementation in practice	30
6.4.1	Input manager	30
6.4.2	Recognition module	32
6.4.3	Warning module	36
7	Performance analysis of PocketSphinx	38
7.1	Benchmarking process	38
7.2	Out of the box configuration	38
7.3	Using a custom grammar	40
7.4	Using a custom acoustic model	40
7.5	Command line arguments	42
7.6	Is PocketSphinx usable for coding with the voice ?	43
8	Conclusion	44
A	Command line arguments of PocketSphinx	46
B	Grammars used	48
B.1	The full python grammar in JSGF form	48
B.2	The full python grammar in FSG form	53
B.3	The simplified python grammar in JSGF form	54
C	International Phonetic Alphabet	56

Chapter 1

Introduction

Voice assistants are becoming more and more popular. Every platform now tries to have a voice-controlled assistant, and every big company has its assistant, like Alexa, Bixby or google assistant. With the rise of those assistants, users are increasingly accustomed to do basic tasks by voice, and voice recognition frameworks are released frequently.

As a large fraction of people work all day on their computer, some injuries/handicap becomes common: if you suffer from RSI (repeatable strain injury), you won't be able to type on a keyboard for long periods; if you broke your arm, your typing speed will decrease dramatically; all those problems affect the abilities of those people to work, and are caused by the input method, the keyboard.

So the rise of the new input method, the voice, is an incredible thing for people unable to use a keyboard. It is now possible to control a computer with the voice, dictate long text and to interact vocally with programs, with a quick setup. But in all cases, they rely on the fact that the text dictated and the command spoken are sentences that have meaning for a native speaker, thus all those voice recognition programs are created and trained to recognize classic text. And so, at the present, programming with voice does not benefit from such progresses.

At the moment, there is no easy solution for dictating code. There are only workarounds, solutions that try to allow coding with the voice, but are only tailored for one specific individual, need a big learning curve with strange keywords to control the dictation, and are not easily usable for a lot of users.

Since code is so much different from continuous text, we cannot just use classic text recognition and call it a day: we must adapt an existing solution and change it to have a good recognition quality.

This thesis will look into this problem; we will see what are the available solutions, what algorithms are used, and see whether they can be used to dictate code by benchmarking their quality in this context.

This work focuses mainly on recognizing code in the browser: This was intended to be an Ingenious plugin, allowing students to complete small samples of code in the browser. Since a browser-based solution is also very easy to adapt for most platforms and applications (since a portion of them are based on Electron [1] or include web rendering), this allows having a portable solution. And since javascript has really good network capabilities, this allows using server-based solutions more easily.

Chapter 2

Voice recognition for the browsers

This chapter explores different solutions to support voice recognition in the browser. The requirement of being available in the browser constrains the solutions to be portable, and usable by any application that can emulate a browser, which is a task that many languages have built-in functions for or many frameworks allowing to render webpages. Since a lot of applications nowadays are built with electron or include some sort of web rendering (such as Slack, Atom, Steam, Whatsapp, etc. [1]) the integration won't even require more work.

2.1 Web Speech API

The first direction to build a voice recognition for code recognition is to use the built-in voice recognition furnished by the browser, as drafted by the W3 consortium [2].

The Web Speech API is not the most extensive one, but it has some important features: you can add a custom grammar, make continuous recognition, and multiple recognition results with an accuracy percentage. With those features, it is possible to build a quite complex voice recognition engine. But like all good features of the web, it depends heavily on the implementation of each major browser, and since it is still a draft W3C and not a standard, the browsers are not necessarily implementing it. Therefore we must check how well it is implemented in each browser.

2.1.1 Google Chrome

Since Google Chrome is the most used browser (with more than 60% of usage across all platforms) [3] [4], it is logical to try to first create a solution that would work on Chrome.

```

var SpeechRecognition = window.SpeechRecognition || webkitSpeechRecognition;
var recognition = new SpeechRecognition();

recognition.onresult = (event) => {
  console.log(event.results[0][0].transcript);
}
recognition.start();

```

Figure 2.1: Minimal code for speech recognition

At first sight, the API seems to be well implemented on Chrome, with a few lines of code being enough to recognize English sentences, as shown in figure 2.1. But once we try to use the more advanced features, such as a custom grammar, the browser ignores them. This comes from the fact that the browser just sends the recording to google servers, that recognize it and return the results, but they do not use the custom grammar provided. This creates good recognition for standard speech but is annoying in our case, since a custom grammar would be really useful. But we will check to see if we can make it work for code even without grammar.

This makes Google Chrome not perfect for code recognition. What about the other competitors of Google Chrome?

2.1.2 Firefox

On Firefox, there is a flag in the about:config, that should enable speech recognition, the *media.webspeech.recognition.enable* flag. So when enabled, everything should work, right? Well, no.

When enabled, and even in the nightly release of Firefox (tested on the 15/05/2019 and on the 28/12/2019), all the setup is working, adding grammars looks like it worked, but the browser cannot start the recognition. Even with all the different configurations and modifications possible to the code, the recognition does not start. This was expected, as even the Mozilla developer website notes that Firefox does not support this feature [5]. It is quite interesting to see that they have a tutorial on voice recognition that their navigator does not support. But the fact that even when activating the flag the functionality does not work is surprising, meaning that the flag is useless at the moment.

2.1.3 Other browsers

The other major desktop browsers, Safari and Microsoft Edge, are pretty clear on the fact that this is not supported yet. This makes those browsers unfit for code

recognition at the moment.

Another big yet relatively unknown browser is Chromium: this browser is the open-source project on which Google Chrome is based: all of google chrome is based on Chromium, and many smaller browsers are based on it. Even Microsoft Edge has moved from its proprietary rendering engine to a Chromium-based implementation [6] [7].

And so, most of the smaller browsers, like Brave or Opera, are functionally the same as Google Chrome, since they are based on the same open-source project and implementation, Chromium. Thus they are not fit to use the Web Speech API in our case. Sometimes they even deactivate the voice recognition functionality for privacy concern (since the speech is sent to Google servers, and protecting privacy is one of the marketing arguments of brave [8] [9])

So none of the other browsers are able, at the time of this writing, to correctly use the web speech API.

2.2 PocketSphinx

PocketSphinx is an open-source speech recognition engine, specifically made for mobile devices, which can also be used on the desktop. This engine has the great advantage of being light-weight, which is necessary for browser usage. It also has a lot of configurable parameters, such as the dictionary used, the grammars or the acoustic models. This is perfect for what we want to build since we will be able to test different features to find which one works best. [10]

2.2.1 PocketSphinx in a browser

However, PocketSphinx is written in C for performance reason, therefore we must adapt it to use it in the browser . Luckily, with WebAssembly [11], it is possible to run C code, compiled in a specific way, in the browser, with near-native performances, allowing us to use this engine to its full power. And even better, an open-source wrapper [12] exists, which allows us to use it without having to recompile it and create the layer of communication between the javascript application and the WebAssembly code. One such implementation and the one used to test this solution is pocketSphinx.js. [12] [13]

This will be discussed more in-depth in chapters 5 and 6.

2.3 Google Cloud Speech, Amazon Transcribe, Wit.Ai, etc.

One of the most popular solutions to do voice recognition these days is to use an external service, such as Google Cloud Speech or Wit.Ai. The advantages of those solutions are the simplicity of using them, lifting the heavy work (doing the recognition) on an external server to avoid computation cost, and to use the quality recognition, thanks to the massive datasets those services usually have.

The common problem with those services is that they do not allow the user to use a custom grammar. At best, they allow to indicate some words that would be more frequently used or basic commands, but this is not as powerful as a complete grammar or a custom dictionary. This makes those services hard to use for accurate code recognition.

They are also closed-source services, with no way of knowing what they do with the data sent, or any way to be sure the service will continue: if the company decides to stop or goes bankrupt, there will be no possibility to recreate the service on personal servers.

There is also another problem with some of these services: their cost. Most of them provide the first hour of recognition free, but after that, they bill at the minute recognized.

This makes these solutions unfit for a large scale use, or non-paying users.

2.4 External server

Another solution is to use a custom external server with custom recognition software. This would allow using a powerful server, leveraging the hard work from the unknown and potentially slow machine of the user to our known and powerful server. This also increases the compatibility across browsers, since the browser just needs access to the microphone and to communicate with a web server, which is quite basic and should be supported by any decent browser (even internet explorer). One of the most used solutions is Dragon, a voice recognition software built by Nuance. There are also some open-source projects, such as Kaldi.

2.4.1 Dragon

Dragon is a speech recognition software created by Nuance. It is one of the most well-known and used voice recognition solution used, with more than 22 million users [14].

Dragon is a powerful tool and recognized as the best solution existing at the moment, but with some downsides. The first downside is that it is not an open-

source program, with all the implications for maintenance and user privacy that come with closed-source programs. The other downside is that it is tailored for a single user, which makes it more difficult to adapt for multiple users, as in the case of Ingenious. One smaller downside is that it is quite costly, with a single person license costing hundreds of dollars, with the mass license being more expensive. But this cost does not increase with the use, thus making it more usable for a mass-scale usage than the paying cloud solutions discussed earlier.

For those reasons, Dragon could be a good choice, but only if the other choices do not yield good enough results.

2.4.2 Kaldi

Kaldi is an open-source project written in C++, that aims to have modern and flexible code [15].

It is a relatively new project that is not as mature as some other recognition engines, but it is quite promising, with some members of the CMUsphinx project (the project which Pocketsphinx is a part of) moving from sphinx to Kaldi [16]. But as it is relatively new, tools and documentation are not as extensive and mature as those of CMUsphinx for example, which makes it harder to use for the moment.

There is mention of a WebAssembly build on the internet, but it has since disappeared. There is also an extension with Gstream that allows it to use a server and a client in a web browser.

2.5 Which voice recognition engine to use

As we can see, there is a large choice of projects to use for voice recognition, but most of them are not open-source. So, for our implementation, we will first try to use the web speech API. Then, we will use, benchmark and analyze PocketSphinx.

Chapter 3

Algorithms for voice recognition

This chapter explores the different algorithms and data structures used in voice recognition. It focuses on sphinx's recognition, since it is easy to apprehend, and it is the basis for many speech recognition engines. We also describe more recent algorithms, without getting too much into the details, since that would require a long time to explain them thoroughly and correctly.

3.1 Voice recognition process

How does sphinx uses those algorithm to recognize sound and speech? Sphinx's recognition works in four steps (one clustering and three passes) :

- first the input will be cut in 25.6ms audio section that will be clustered using the GMM and phonetic fast match into phonemes.
- then a Viterbi beam search pass in the forward direction (from the beginning to the end of the utterance), using the full vocabulary. The result of that search will be a single sentence (the most probable), and a word lattice that will contain all the words recognized during the search.
- after that, a Viterbi beam search pass in the backward direction (from the end of the utterance to the beginning), using only the words in the word lattice previously found. This pass will create multiple hypotheses, and will also produce the best path score from any point in the utterance to the end, which will be used in the third pass.
- and finally, an A* search using the word lattice and score produced by the two first passes, which will create a list of the best hypothesis

After that, the best hypothesis will be selected as the recognized sentence. This process is schematised in the figure 3.1

The reason to use three passes instead of just using the first (which already produce a hypothesis) is for performance reason: the first pass use simple model (bigram models) to reduce the search space of the next passes, that will be more precise and use the full model. Since they will have a smaller search space this will allow real-time recognition.

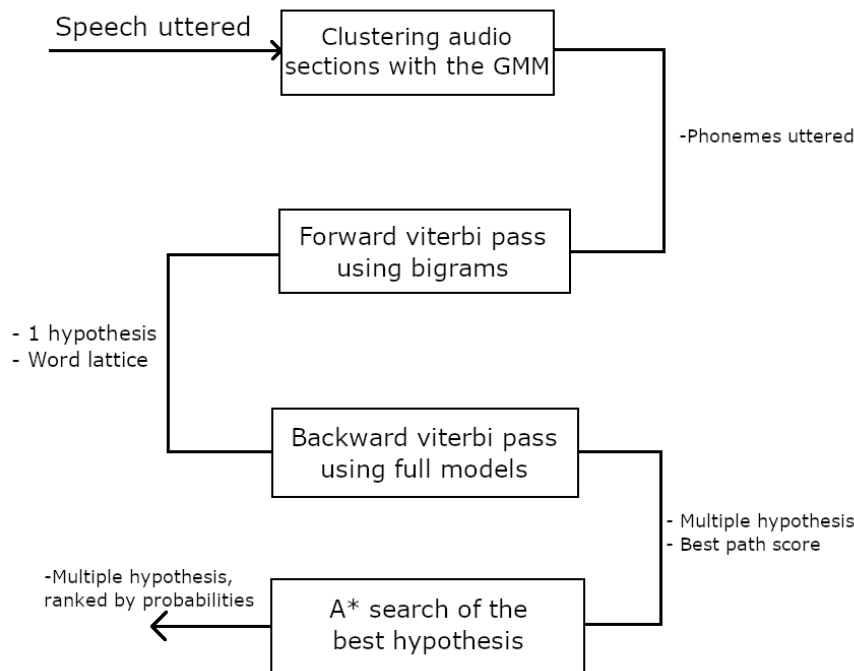


Figure 3.1: The voice recognition process used by Sphinx

3.2 The Viterbi algorithm

The Viterbi algorithm [17] is an algorithm used to find, in a hidden Markov model, the most likely sequence of states that produces an observed stream of events.

The algorithm works by creating two matrices, one that keeps track of the best probability of being at a given state, and the other the path that was taken to reach that state. At each step, the algorithm calculates for each state the probability to reach this state with the correct event, and remember the best score and the

path associated. Then, once the algorithm as made as many steps as the number of events observed, you just need to take the most probable state of the last step, and the associated path is the most likely path to create this series of events.

The Viterbi algorithm takes in argument :

- the number of possible observations O
- the number of states K
- the starting chances Π (of size K): the chances to start at a specific state
- the sequence observed Y : the observations for which we must compute the most likely path
- the number of observations N
- the transition matrix A (of size $K \times K$): this matrix indicate the chances of transition from one state to another
- the emission matrix B (of size $K \times O$): this matrix indicate the chances of emitting a specific observation when leaving a state

A pseudo-code for this algorithm is shown below :

```
function viterbi(O, K, pi, Y, N, A, B) :
    var paths = matrix(N+1, K)
    var probabilities = matrix(N+1, K)

    for(i = 0; i < K; i++) :
        probabilities[0][i] = pi[i]
        paths[0][i] = {}

    for i in range(1, N+1) :
        for j in range(0, K) :
            var best = -1
            var bestScore = 0
            for w in range(0, K) :
                if w != j :
                    var newScore = probabilities[i-1][w] * A[j][w] * B[j][Y[i-1]]
                    if newScore > bestScore :
                        best = w
                        bestScore = newScore
            probabilities[i][j] = bestScore
```

```

paths[i][j] = paths[i-1][best]+best

return max(probabilities[N])

```

Of course, this algorithm can be modified according to the need, with the emission matrix linked to a transition instead of states, the starting chances not specified (in which case either a starting state is specified or all the state have the same start chance), or other changes.

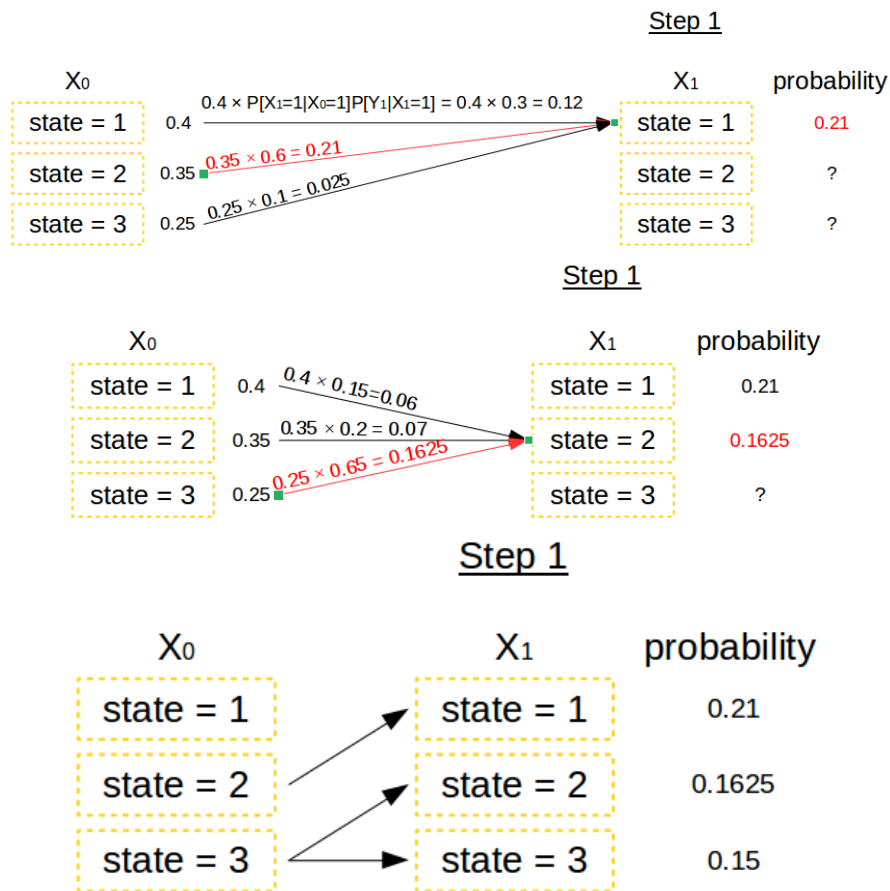


Figure 3.2: a visual representation of the Viterbi algorithm
<https://medium.com/analytics-vidhya/viterbi-algorithm-for-prediction-with-hmm-part-3-of-the-hmm-series-6466ce2f5dc6>

3.3 Beam search

The Beam search is a heuristic that can be applied to search algorithms (and mainly to the Viterbi search). With this heuristic, instead of calculating and remembering all the probabilities for each state at each step, the less probable results are discarded. The discarded results are determined by the argument beamwidth and are the results that have a score lower than the best result's score multiplied by the beamwidth (which is between 0 and 1).

This is a greedy optimization that works well in practice but one that needs to be fine-tuned to find the best value for the beamwidth that will determine how many results are discarded [18] [19].

If beamwidth is too close to 1, too many results will be discarded and the quality will degrade, and if beamwidth is too low (close to 0), no result will be discarded and the heuristic will be useless. So it is a balance to strike between quality and performance

3.4 A*

The A* (A star) algorithm is a graph traversal and path search algorithm. It allows finding the shortest path in a graph if the chosen heuristic function is admissible. It is a widely used path search algorithm, having better performance than a Dijkstra search while being as optimal with the correct heuristic.

The algorithm works with a priority queue, with the starting node in it. For each adjacent node, the algorithm attributes a score, which is equal to the cost to get to that node plus the estimated cost to reach the goal from that node. The estimated cost is the heuristic. Then the algorithm expands the next node with the smaller score (so the node with the lowest cost of reaching it plus the estimation of reaching the goal) and adds its neighbor to the priority queue (and also stores the path to those neighbor). It continues until it reaches the goal, and then it has to take the path stored to find the smallest path from the start to the goal.

Here is a pseudo code implementation of the A* algorithm :

```
function heuristic(node, goal) :  
    var distX = node.x-goal.x  
    var distY = node.y-goal.y  
    return sqrt(distX*distX+distY*distY)  
    // A simple heuristic;  
    // it estimate the distance as a straight line to the goal  
  
function astar(start, goal) :
```

```

queue = priority_queue()

// Specify that the queue will sort element by their cost
queue.setSort("cost")
queue.add({cost : 0, score : 0, node : start, path : {}})

while ! queue.empty() :
    var element = queue.pop()
    if element.node == goal :
        return element.path
    foreach element.node.neighbor as neigh :
        queue.add({
            cost : element.node.cost + neigh.cost,
            score : element.node.cost + neigh.cost + heuristic(neigh, goal),
            node : neigh.node,
            path : element.node.path + neigh.node
        })

return "the goal is unreachable"

```

In this example, each node has coordinates (x, y), and a list of neighbors to which they can go, which is composed of a cost (the cost to go from the node to the neighbor), and the node in itself. The priority queue sorts the element by their score, from the lowest to the highest (so that `queue.pop` will return and remove the lowest score of the queue). Unless the cost to go from one neighbor to another is lower than the straight line between those two, the algorithm will find the best path.

This algorithm finds the optimal path if the heuristic h is monotone (= consistent). A heuristic is monotone if its estimate is always less than or equal to the estimated distance from any neighboring vertex to the goal plus the cost of reaching that neighbor. So the algorithm is optimal if the heuristic never overestimates the cost.

A classic heuristic in a map is the straight-line distance, since it is always the smallest possible distance between two points, and it is monotone.

We can also note that choosing the heuristic $h(n) = 0$ will lead to the same execution as a Dijkstra algorithm. (since it expands the nodes closer to the start, without preferring to extend those close to the goal)

3.5 GMM

GMM are Gaussian Mixture Models, a subtype of Mixture Models. Gaussian Mixture Models are used in machine learning to cluster data while allowing an element in one cluster to have probabilities to be in another cluster (soft-clustering method). So each element will have a probability to belong to one cluster, and not just belong to only one cluster, like with the K-means cluster method, which is a hard-clustering method.

A Gaussian mixture model is comprised of multiple Gaussian, with three parameters that define the Gaussian; the mean μ , the covariance σ and the mixing probability π . Then to cluster new data, you just check in which Gaussian it is. This is visually shown in figure 3.3 (in 1 dimension, but this is expendable in multiple dimensions)

The GMMs are used with the language model, to recognize the phonemes.

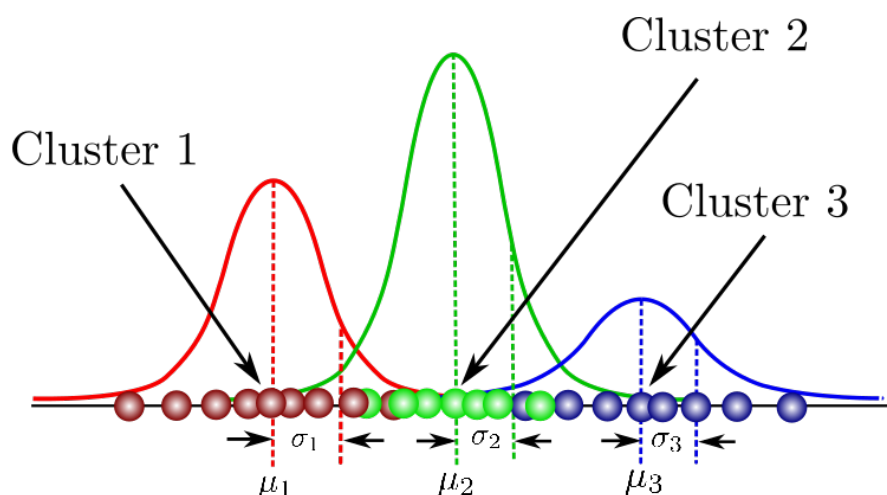


Figure 3.3: Visual representation of the clustering using GMM

<https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95>

3.6 SGMM

SGMM, Subspace Gaussian Mixture Models, is an improvement of the classical GMM specifically for the voice and speech recognition. In this variation, all the phonetic states (all the state of the HMM) share the same GMM, but the means and mixing probability will vary in a subspace according to the active phonetic state.

3.7 Phonetic fast match

The phonetic fast match is a heuristic used in the phoneme recognition, that will compute phonemes in future frames instead of just the active one. It will then calculate the possible phonemes candidates for the next frames, which will reduce the possible candidates and will speed up the recognition to find the correct phoneme. This combined with a beamwidth (that works just like in beam search), will reduce the search space, but can also reduce the quality of the search if the arguments are not fine-tuned.

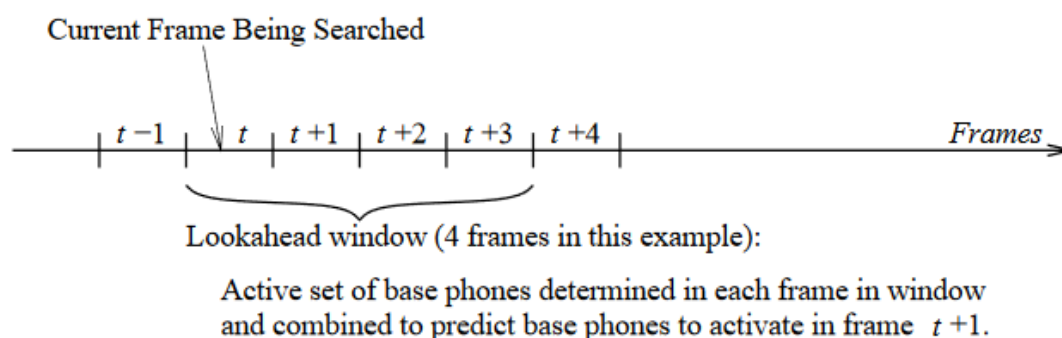


Figure 3.4: Visual representation of the PFM
Source : Efficient Algorithms for Speech Recognition

3.8 Other algorithms and more

There are many other algorithms used by recent speech recognition engines, such as boosted MMI [20], FST weight-pushing [21], but those algorithms are more complicated, would be too long to explain in this work, and most of them, for the moment, are only used in experimental speech recognition engine.

If the reader is interested in knowing more about this subject, the Ph.D. thesis "Efficient Algorithms for Speech Recognition" by Mosur k. Ravishankar is an excellent read. Even though it is old (more than 20 years ago), its content is still relevant today and a good introduction to speech recognition.

The documentation of the Kaldi project contains also a lot of information on the algorithms used [15], and their research paper is also very instructing [21].

A lot of other information can also be found in machine and deep learning papers, as this field heavily influences speech recognition. Most of the advances in those fields contribute to speech recognition, with things such as SGMM, boosted MMI, MCE discriminative training and many others.

Chapter 4

Voice recognition for code

When creating voice recognition for code, there are many parameters to take into account, lots of which are different than when recognizing classic speech. Code is very different from classic spoken English, even if there are some resemblance. This allows us to build a speech recognition for code using the advances of modern speech recognizer, but a lot of modifications and adaptation will have to be made.

4.1 Voice recognizer are built for sentences

Most of the research on speech recognition is made to recognize usual spoken human speech. Since most speech recognizers are used to write down discourses or to allow another form of input where the user dictates the command to the computer, this is perfectly logical. The continuous speech is built in sentences, that have little connection between them; for the recognition, it is not important to understand to which person the pronoun "her" refers to, it is not even important to check if it could refer to a previous character if we have enough accuracy to assume it was what the speaker said. It could be used in a super-accurate speech recognizer, but there is the problem of trying to interpret the potentials errors of the speaker, which would then reduce the accuracy of speech recognition. Thus, the speech recognizer usually only tries to recognize each sentence separately and does not try for the most to analyze the meaning of the text. So the recognizer uses the context inside the sentence to increase accuracy, but not the context of the whole discourse.

But for code, it is quite different, the context is very important. There are not many keywords, but they can be anything, since variables and function names can usually be any ASCII character, and are usually used differently according to their type. So the context of the whole code is important to have a good accuracy. One

variable and one function can have very similar names, but should not be used in the same way.

4.2 Code have multiple context

In a discourse, the different sentences are independent. There is no "illegal" sentence that cannot come after another. At most, there are illegal words in a sentence, but no rules between the sentences. But for code, there are usually some rules that depend on the context. In some language, you cannot declare a function inside a function. You cannot use *else* if there was not an *if* before, etc. Using the context, we can determine which lines are most probable, and which lines won't be used because really improbable or illegal (declaring `#include` in c in the middle of a file is legal, but is rarely used).

This allows to greatly reduce the possible sentences at a given time, and also allows recognizing more easily the variables and functions, since there is a high probability that they were previously declared.

4.3 Existing projects

Programming by voice is a subject that hasn't gotten a lot of traction, even though there are more and more projects based on voice recognition.

There is a talk of Tavis Rudd at PyCon [22] where he shows his solution for voice coding, which includes a full language of commands to control the cursor and other text manipulation, which is hard to learn and lead to saying sentences such as "pam snake my dictionary ak par". This is not ideal for a novice user. It requires a learning period, and even after a long time, it is still not intuitive. So even though it allows for complete control, this is not ideal. This approach is the most common, using keywords to control the cursor, and recognizing word by word. The problem is that each user has to learn the keywords, and none of the specifications of the code is used. The grammar of the language is not used, no context is used, and this approach could be used to edit any kind of text.

There are some other projects, such as Voicecode [23] or Vocola [24], but those solutions go the same path as Rudd's solution and are constrained to Windows. So all these solutions are good to use if you know you will need to code for a long time with voice and you have the time and motivation to learn to use them, but for an Ingenious plugin, this is not usable.

4.4 Conclusion

Programming with the voice is a special case of voice recognition, that can take advantage of those specifications to improve accuracy. But at the moment, most of the projects to recognize code are hard to learn, do not use those specificities and even though they work, are only suitable for specific cases (one user motivated enough to learn it or create his/her keywords)

Chapter 5

PocketSphinx

PocketSphinx is a large project. It offers a lot of options and parameters to fine-tune the recognition. In this chapter, each of those parameters is explained in detail, and in the next chapter, those parameters are benchmarked to precisely measure their impact on recognition accuracy.

5.1 Grammar

The grammar is an important part of voice recognition or text parsing. It allows us to specify what words are usable at each moment, and to easily parse, understand and label text.

There are two ways to specify a grammar in PocketSphinx: using a java speech grammar format (JSGF) file or with a finite state grammar (FSG)

5.1.1 JSGF

The JSpeech Grammar Format (JSGF) is a textual representation of a grammar to be used for speech recognition. The fact that it was created with voice recognition in mind makes it interesting, because it adds a few features that are useful, such as tagging or weights. [25] [26] [27]

This grammar supports tokens, rules and block controls. But there is also a possibility to define the weight of an alternative, to indicate the likelihood of a choice being taken. The grammar also allows tagging block, allowing to parse more easily a text (or a spoken discourse).

Defining rules

To define or to reference a rule, it must be enclosed with `<>`. For example, to define the rule "greetings", it should look like this :

```
<greetings> = hi
```

Then, inside a rule, you can use control blocks to indicate choice or repetition. The parentheses define a group of word. The pipe operator `|` indicates choice inside a group, and `[]` indicates an optional block. For example, the rule

```
<greetings> = (hi | hello) Kevin [I hope you have a beautiful day]
```

can validate the sentences "hi Kevin", "hello Kevin", "hi Kevin I hope you have a beautiful day" or "hello Kevin I hope you have a beautiful day".

Other operators are the Kleene star, that indicates the group may be spoken zero or more times. Thus the rules

```
<please> = (please | oh please)
<asking> = <please>* don't crash
```

validate the sentences "don't crash", "please don't crash", "oh please please don't crash" or even "oh please oh please please don't crash" (and many more). There is also the plus operator `+`, that does the same thing, except that the group needs to be spoken at least one.

Tags and weight

Tags are used to add more information to a group that can be used by the application. They are defined with `{ }` and apply to the group just behind them. they do not change the sentences recognized, but they help its interpretation. So the rule

```
<greetings> = (hi | hello) Kevin {NAME} [I hope you have a beautiful day]
```

is not different than the one described earlier, but the token Kevin, when recognized with that rule, will have the tag NAME. This allows, for example, to translate the grammar, with the tags staying the same and the token changing according to the language.

You can also add weight to groups using `/ /`, to indicate which group is most likely to appear and thus improve the quality of recognition. For example, in

```
<greetings> = ( /10/ hi | /1/ hello) Kevin
```

The sentence "hi Kevin" is ten times more likely to appear than "hello Kevin". This allows specifying rarely used sentences without reducing the accuracy of frequently used sentences. Note that the weight can be a floating number, like `/0.034/` or `/4.429e2/`

Python grammar in JSGF form

To use JSGF with python recognition, we must transform the python grammar defined in the python specification [28]. This gives us the grammar in Appendix B. This is quite a big grammar, with for example the rules for listing the arguments of a function taking 12 lines. The grammar separate the differents statements into small and compound statements. The simple statements are break, continue, assert, or other small lines, and the compound statements are longer instructions, such as if, for, function definition or other long statements.

5.1.2 FSG

The second way to describe a grammar to PocketSphinx is with a finite state grammar. This grammar is defined by multiple states, and transitions that go from one state to another, with a probability attached to it.

To describe such grammar, one must first describe the transitions, their source, their destinations, the word that triggers the transitions, and their weight. In pocketSphinx.js, this looks like

```
var transitions = new Module.VectorTransitions();
//logp is the logarithmic probability,
//so logp=0 means that the probability is 1
transitions.push_back({from: 0, to: 1, logp: 0, word: "HELLO"});
transitions.push_back({from: 1, to: 2, logp: 0, word: "WORLD"});
```

Once the transitions are defined, we only need to add the grammar to PocketSphinx. In PocketSphinx.js this can be done by calling the function addGrammar of the recognizer object

```
recognizer.addGrammar({start: 1,
                      end: 2,
                      numStates: 3,
                      transitions: transitions})
```

This creates the 3 states with the transitions and sets the state 1 as the beginning state, and state 2 as the ending state. (multiple states can be defined as ending states)

These two ways to describe grammars have both their advantages and disadvantages. JSGF is easier to create and read by a human, but FSG is lower-level and is the format that will be used by PocketSphinx in the end, since the JSGF file will be converted in FSG at runtime. But since it is easier to convert JSGF to FSG than the reverse, we usually work with JSGF. The FSG corresponding to the python grammar is in Appendix B.

5.2 CMU pronouncing dictionary

To describe how words are pronounced, we usually use the IPA, International Phonetic Alphabet. This alphabet is useful to describe pronunciation from all languages, but the problem is that this alphabet uses more symbols than the English language, thus making it hard to use in code or to type with a keyboard. This is why projects to translate the IPA into ASCII characters exist, to allow using it more easily. The CMU pronouncing dictionary is one of those. In Appendix C, we provide the correspondence of the pronouncing dictionary with the IPA, and in the table 5.1 you can see a few examples [29] [30] [31] [32].

CMU form	IPA form	CMU form	IPA form
AA	ɑ	B	b
EY	eɪ	CH	tʃ

Table 5.1: Examples of the CMU Pronouncing Dictionary

5.3 Web worker and Wasm

At its base, Javascript is mono threaded but event-based. This means that when performing long calculations, it either blocks the UI thread or is interrupted frequently. Luckily there is a way to add multithreading: WebWorkers. WebWorkers are small programs that can be launched from the main script, and that communicate with it by message passing. This allows to free up the main thread while doing continuous, CPU intensive task (and using the multiple cores of modern processors). [33]

But to achieve good performance with CPU intensive tasks, we must use Wasm. Wasm is short for WebAssembly and is a binary instruction format to be used by the browser (or any Wasm VM), to achieve near-native application performance. It is compiled from low-level languages, like C++ or rust, and can achieve performance very close to programs compiled directly from those languages. This also allows to easily port existing C/C++ application to the web. [34]

So for PocketSphinx.js, there is both a WebWorker and a Wasm program: the Wasm performs the recognition in itself, since this is the most CPU intensive task, and the WebWorker handles the recording, using the high level, easy to use javascript API to access the microphone, and then sends the data to the Wasm program.

5.4 Acoustic model

PocketSphinx works with acoustic models. These are files that represent the relationship between the audio and the phonemes described earlier. They are necessary to allow recognition, and good or bad acoustic model can make the difference between good and bad recognition. It is also important to note that acoustic models are adapted to a context: an acoustic model trained with a 1000 euro microphone will be quite different from an acoustic model trained in a noisy room with a laptop microphone, so it is very important to take acoustic models adapted to the average use-case (and to have determined such average user).

There are two ways to increase the accuracy of the acoustic model: adapting a high-quality existing one or training a custom one. Those two options are discussed and benchmarked in the next chapter. [35]

5.5 Command line arguments

There are also a lot of command line arguments that can be used. Some of them control the behavior of the recognition engine, some allow to balance the performances and quality, and some allow to specify external files to be used. They are described in Appendix A, and will also be discussed in the chapter 6.

Chapter 6

Our implementation

6.1 A modular library

The goal while creating this program was to make it modular and versatile. Each part of the program should be in its independent module, to allow to easily change the voice recognition engine, the input manager or the code formatter.

To do so, each part of the software lives in its file and is loaded at runtime. This allows loading only the modules needed and to potentially change them at any moment. Once they are loaded, they communicate with each other using callbacks and registering for events. Each module of the same type has the same interface, allowing to change one module with only one line of code. This also allows anyone to modify part of the library to tailor to their needs without having to understand all the code or to change more than they need to.

This also allows to easily test one module, by just checking its entry points, just like testing an API.

This project is a jQuery library, to be easily used in other projects, and to ensure compatibility with the jQuery function the project used. [36]

6.1.1 The recognition

The recognition process is organized in two phases. The structuring phase and the naming phase. During the structuring phase the user will dictate the structure of the code, using a specific set of keywords allowing to easily dictate and precisely recognize the python code, but with all the class/function/variables names and string replaced by a keyword (className, function, variable and verbatim respectively). Then the second phase, the naming, allows the user to replace those four keywords by the names he/she will have decided. This allows us to increase the quality of the recognition by not trying to recognize complicated custom variable

names and would allow the user to type the variable name with a keyboard if they are too complex to dictate or to enable letter by letter dictation. This also avoids having to change grammar during the recognition, allowing instead to use one grammar for the code structure recognition, and keeps the flow of dictation the same for the user, since he/she will not have to stop and dictate letter by letter, instead focusing solely on the structure by dictating a set of keywords.

6.1.2 The modules

Our library is divided into the following modules :

- the main module: the module that loads the other modules, and once all the modules are correctly loaded, launches the input manager module.
- the input manager module : This module makes the link between the HTML element `<textarea>` or `<text type="input">` and the voice recognition. This module adds the button to start the recognition, and transforms the recognized text into real code, and manages the transition between the phases of code structure and verbatim text.
- the warning module: This module prints a tutorial for the user. The first time the user tries to use the voice recognition, a popup will show, explaining how the voice recognition works, which keywords to use, etc., and also asking for consent to use the data for research, and to send the voice to an external server (if applicable).
- the recognition module: the most important module, the one that performs the actual voice recognition. This module is responsible to initialize the microphone, recognizing the utterance, and sending regular hypotheses to the input manager. This module can do the recognition in local (like PocketSphinx), or send it to an external server (like WebSpeech on Chrome or Kaldi), as long as it sends regular hypotheses.

The main module

The main module is the one that creates the jQuery module and loads all the other modules with the jQuery function *getScript*. Once all the scripts are correctly loaded, the module initializes the recognition module (and specifies if it is for testing or not). It then initializes the input manager for each element passed to the jQuery module.

This module has no callback or entry point for the other modules, as it is called by the user itself.

The recognition module

The recognition module is the core module that performs the actual voice recognition. It exposes 5 functions.

- the function `start`: this function takes a callback as argument, and starts the recognition. Whenever there is a result available, the callback is called with the data of the recognition thus far.
- the function `stop`: this function simply stops the recognition.
- the function `onReady`: this function takes a callback as an argument that will be called once the recognizer is ready. If the recognizer is ready when the function is called, the callback will be called immediately. This function must be called before the function `start` to make sure that the recognizer is ready, because if the recognizer is not ready it may result in an error.
- the function `onEnd`: this function takes a callback as an argument that will be called when the recognizer finishes recognizing. Most of the time this will be after a call to `stop`, but sometimes it could be due to an error or crash.
- the function `restart`: this function takes one argument, the phase (either "structuring" or "naming", as described earlier). This function restarts the recognizer, and changes its phase, either "structuring" or "naming" (by default, the recognizer will be in the structuring phase when starting). This allows us to change the grammar of the recognizer according to what should be recognized, and will not trigger the "onEnd" events (that where specified by the `onEnd` function).

When the module is loaded, the webworker is started, the acoustic models, dictionary, and grammars are loaded. Then the media devices (microphone, webcam, etc.) of the user are scanned, and the permission to use the audio will be requested. Once all of this is done, the "onReady" events (specified by the function `onReady`) are fired. When the recognition in itself starts (with the `start` function), the recording starts using the grammar and dictionary according to the mode, and the hypothesis will be emitted to the callback provided in the `start` function.

The warning module

The warning module exposes one function: `check`. This function takes two arguments, the callback when the warning is accepted, and the callback when it is refused. When this function is called, a panel will be added to the page's HTML, with an explanation for the user, and buttons to allow him/her to accept or refuse.

Once the user has made a choice, the corresponding callback will be called, and a cookie will be stored, to avoid prompting him/her again, and immediately call the corresponding callback next time.

The input manager module

The input manager module exposes one function: `init`. This function initializes the HTML element `textarea` or text input passed in argument, to make it ready for recognition. To do so, it adds a callback to the recognition module using the function `onReady`, so that once the recognition module is ready, a button shows up on the `textarea` element. Once this button is clicked another callback will launch the warning check function, and once the warning is accepted, the recognition can start. If the recognition was already started, the button simply stops the recognition.

In the recognition, the two phases described earlier take place. During the structuring phase, the module simply replaces some specific keywords with the corresponding symbols. For example, `"twoPoints"` gives `":"`, `"addTab"` adds a tabulation and `"endLine"` finishes the line and adds a line return. This phase ends when the keyword `"endText"` is recognized. Once this phase is finished, the second phase, the naming phase, starts, and the module selects each keyword that needs to be replaced one at a time and will use a general-purpose recognizer to recognize the sentence the user is saying. Once the sentence is finished, or one of the finishing keywords is recognized (`"end"`, `"finish"` and their variations), the sentence is transformed into camel case, the keyword is swapped with the sentence, and the process begins with the next keyword that needs to be changed. Once each of those keywords is replaced, the recognition is finished.

6.2 Web speech API

The first attempt for the voice recognition module was made with the web speech API. In Google Chrome, the Web Speech API works by directly sending the audio to the Google server, and we cannot use a custom grammar. For the naming part this worked well (since it is classic speech recognition, something google is quite good at), but for the structuring part, it was not working well. As expected, the speech recognized was too heavily modified to correctly guess the code uttered. Using the Levenshtein distance [37] (the minimum number of insertion, deletion or substitution to make to a string to transform it into another string), we can correct some of the words, but long keywords are split into smaller keywords when google tries to interpret the speech, and this makes it not possible to recreate the code uttered. One example of those results is shown in table 6.1

Text uttered	recognized by google	corrected
if	all	and
openParenthesis	does this close	def elif global
variable	variable	variable
equal	equal	equal
variable	valuable	variable
colon	colon	colon
closeParenthesis	current EST	continue as
endText	and text	and endText

Table 6.1: recognition using Web Speech API

6.3 PocketSphinx.js

The second attempt for the voice recognition module was made with PocketSphinx. PocketSphinx is a part of the CMUSphinx open source project, a project developed by Carnegie Mellon University. [38] The CMUSphinx project is quite old, with the first version of sphinx released in 1986, and PocketSphinx released in 2013.

The module was created with PocketSphinx.js, a WebAssembly build of PocketSphinx, made for the browser. The results of this implementation is discussed in the next chapter.

6.4 The implementation in practice

In this section we walk through the implementation in javascript, with some code example. Of course all the code is not shown, it can be viewed on the project's github [39], as well as all the test samples, tools and other things used in this project.

6.4.1 Input manager

basic.js

This module adds a HTML button to the textarea given in argument, and this button is styled in css to be invisible.

```
var node = document.createElement("button");
node.classList.add("memoireBtn");
$(input).after(node);
```

And once the recognition is ready, the button is set to show it when the cursor is hovering using css and a HTML class (when the mouse goes over the textarea)

```
$.memoire.recognition.onReady(() =>
{
    node.classList.add("memoireBtn-ready");
    // Continue setting up after the recognition is ready
```

after that, when the button is clicked by the user, two things can happen: either the recognition starts or if it was already started, it stops.

```
if(isRecording)
{
    $.memoire.recognition.stop();
}
else
{
    $.memoire.warning.check( () => $.memoire.recognition.start((x) => {
        //recognition started
```

During the recognition, in the first mode (structuring mode), the text is parsed and the keywords replaced (with a mapping of words), and flags are set for the end of lines or text

```
else if(w == "endLine")
{
    flagEnding = 1;
}
else if(w == "endText")
{
    flagEnding = 2;
}
```

Once the flag for endText has been set, we go into the naming phase, in which the recognized text is camel-cased using a regex that matches the whitespaces \s+ and the first letter that comes after (.)

```
function toCamelCase(string){
    return string.replace(/\s+(.)/g, function (match, group) {
        return group.toUpperCase()
    })
}
```

After that, we replace the first instance remaining of a replaceable keyword (function, className, variable) by the camelCased sentence. And when a stopping keyword is recognized (end, stop, etc.) we replace the keyword in the array and we move to the next keyword. Once all keywords are transformed, we stop the recognition

```
if(resultReplace.numberRemaining == 0)
    $.memoire.recognition.stop();
```

6.4.2 Recognition module

browser.js

This module was the first created to test the capabilities of the Web Speech API.

The first version was using a grammar of all the python keywords (for the structuring phase)

```
grammar =
'#JSGF V1.0;' +
' grammar keywords;' +
' public <keywords> = False | class | finally | is | return | None | continue
```

But it was soon discovered that none of the browsers were fit to use it.

The second version used a Levenshtein distance to try to match the word recognized by the Google servers with the possible keywords that were originally said

```
function getEditDistance(a, b){
    if(a.length == 0) return b.length;
    if(b.length == 0) return a.length;
    var matrix = [];
    var i;
    var j;

    for(i = 0; i <= b.length; i++){
        matrix[i] = [i];
    }
    for(j = 0; j <= a.length; j++){
        matrix[0][j] = j;
    }

    for(i = 1; i <= b.length; i++){
```

```

    for(j = 1; j <= a.length; j++){
        if(b.charAt(i-1) == a.charAt(j-1)){
            matrix[i][j] = matrix[i-1][j-1];
        } else {
            matrix[i][j] = Math.min(matrix[i-1][j-1] + 1,
                                    Math.min(matrix[i][j-1] + 1,
                                                matrix[i-1][j] + 1));
        }
    }
}
return matrix[b.length][a.length];
}

```

This allowed us to test the solution. We realize that it was not possible to use this recognition engine since it modifies too much the speech uttered by the user and tries too much to recognize a normal sentence.

sphinx.js

This module works by first spawning a webworker and loading the file needed for the recognition (grammar, language models, dictionary, etc.)

```

spawnWorker("/dist/js/pocketSphinx/recognizer.js", function(worker)
{
    // This is the onmessage function
    // We will receive message from the webworker with it
    worker.onmessage = function(e)
    {
        if (e.data.hasOwnProperty('id'))
        {
            var clb = callbackManager.get(e.data['id']);
            var data = {};
            if ( e.data.hasOwnProperty('data')) data = e.data.data;
            if(clb) clb(data);
        }
        if (e.data.hasOwnProperty('hyp') && !e.data.hasOwnProperty("final"))
            updateHyp(e.data.hyp);
        if (e.data.hasOwnProperty('status') && (e.data.status == "done") &&
            e.data.hasOwnProperty('command') && (e.data.command == "initialize"))
            recognizerReady();
    };
}

```

```

    postRecognizerJob({command: 'lazyLoad',
    data: {folders: ["/", "am"]], files: [
    ["/", "base.jsgf", "../../../grammar/base.jsgf"],
    ["/", "base.dict", "../../../dict/base.dict"],
    ["/am", "means", "../../../am/means"],
    ...
    ["/am", "noisedict", "../../../am/noisedict"]
    ]}
    }, initRecognizer);
});

```

It then scans the mediaDevices of the user, and asks permission to use the microphone

```

if (navigator.mediaDevices.getUserMedia)
{
    navigator.mediaDevices.getUserMedia({audio: true})
    .then(startUserMedia).catch(function(e) {
        updateStatus("No live audio input in this browser");
    });
}

```

With the startUserMedia function creating the media stream and adding it to the array of recorder

```

function startUserMedia(stream)
{
    var input = audioContext.createMediaStreamSource(stream);
    window.firefox_audio_hack = input; // Firefox hack
    //https://support.mozilla.org/en-US/questions/984179
    var audioRecorderConfig = {
        errorCallback: function(x) {console.log("Error from recorder: ",x);},
        worker:"/dist/js/pocketSphinx/audioRecorderWorker.js"
    };
    recorder = new AudioRecorder(input, audioRecorderConfig);
    if (recognizer) recorder.consumers = [recognizer];
    isRecorderReady = true;
};

```

After that, the initialization is complete, and once PocketSphinx is ready, the callbacks registered by onReady (that were stored in an array) are called

```

$.memoire.recognition.onReady = function(event)
{
    if(isRecognizerReady)event();
    else eventsOnReady.push(event);
}
function recognizerReady()
{
    if(isRecognizerReady == true)return;
    isRecognizerReady = true;
    eventsOnReady.forEach(x=>x())
};

```

After that, when the function start is called, the recognition starts, and the function initRecognizerMode sets the parameters of the recognizer, according to the mode (structuring or naming) specified.

```

function initRecognizerMode(mode)
{
    if(mode == "structuring")
        postRecognizerJob({
            command: 'initialize',
            data : [["-jsgf","/base.jsgf"],["-dict","/base.dict"]]
        },
        function(){
            if (recorder) recorder.consumers = [recognizer];
        }
    );
    else
        postRecognizerJob({
            command: 'initialize',
            data : [["-jsgf","/2kWords.jsgf"],["-dict","/2kWords.dict"]]
        },
        function() {
            if (recorder) recorder.consumers = [recognizer];
        }
    );
}

```

This module also includes a "test" argument, that when not null, contains a path to a file to use instead of the microphone. This file is loaded with an audio

HTML element and is transformed into a stream once the audio is loaded and ready

```
if(testing !== null)
{
    audio = new Audio(testing);
    audio.oncanplay = () =>
    {
        const stream = audio.captureStream();
        startUserMedia(stream);
    }
}
```

6.4.3 Warning module

`inginius.js`

This module, created for the beta on Inginious, is quite simple. When the function `check` is called (meaning that the recording button has been pressed), it creates a *div* element, populates it with the HTML of the tutorial, and appends it to the document (and styles it with CSS so it is shown in a fixed position above the rest of the page).

The button created in the HTML also needs to be linked to the `accept` and `refuse` callbacks sent in arguments.

One last element is to manage the cookies to remember whether the user has already read and accepted the tutorial/conditions, to avoid asking him again.

```
function setCookie(cname, cvalue) {
    var d = new Date();
    d.setTime(d.getTime() + (365*24*60*60*1000));
    var expires = "expires="+ d.toUTCString();
    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";
}
```

```
function getCookie(cname) {
    var name = cname + "=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';');
    for(var i = 0; i <ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') {
            c = c.substring(1);
        }
    }
    return c === name ? ca[i+1].trim() : null;
}
```

```
    }  
    if (c.indexOf(name) == 0) {  
        return c.substring(name.length, c.length);  
    }  
    }  
    return "";  
}
```

The first function creates and sets a cookie for a year, and the other reads a specific cookie.

Chapter 7

Performance analysis of PocketSphinx

7.1 Benchmarking process

To make a benchmark of the quality of recognition, it is important to be close to the real conditions of recording and usage, while also being reproducible. To do so, I used a modified version of the recognition module, that takes input from an audio file instead of the microphone. This was possible using the `mediaStream` interface, which is used to record the microphone of a user, but which can also capture the stream from an audio file, which makes it easy to swap from microphone to file.

In this chapter, the quality of PocketSphinx's recognition is discussed. All the audio samples used are available on the GitHub of the project [39]. Each snippet was captured multiple times with different microphones and different speeds to ensure that there was no error linked to a specific recording or dictation speed. They were kept relatively short (about 20 seconds), to reduce the risk of cascading errors (mainly when using a grammar)

This chapter explores all the possibilities to improve PocketSphinx's recognition quality. We test using a grammar, changing the language model, command-line arguments, etc.

7.2 Out of the box configuration

The first thing to try is to use the "out of the box" configuration. This means using the default arguments, with a grammar that does not restrict possible sentences and allows words to be said in any order. This still requires a setup, as the dictionary must be provided, and a dummy grammar created (a grammar that allows any of the words to be entered at any moment).

A few results come from this experiments: With the full dictionary, there is a lot of noise: a lot of "in", "if", and "as" are recognized when nothing should be recognized. Another constatation is that about half the words are recognized, and the other half are mingled with close words, such as "endLine" recognized as "height height" or "if", "elif" and "def" being mixed. An example of that can be seen in tables 7.1 and 7.2, showing the samples 1/2 and 2/3.

The tables 7.1 and 7.2 show the final results broken down with the words that should have been recognized, but during the utterance the hypothesis change; it is not transcribable here, but the experiment can be redone on a computer. This shows that sometimes the computer correctly recognizes the word but changes it after hearing the rest of the sentence.

Text uttered	Text recognized
def	eight
function	function in
openParenthesis	while in elif as
closeParenthesis	as colon in two as
colon	as colon in
endText	in else
colon	colon in
endLine	eight eight

Table 7.1: The recognition of the sample 1/2.mp3, with line break to show which parts of the sentences gave which results. The sentence was said with a clear separation between successive words, but still as continuous sentence

Text uttered	Text recognized
if	function
variable	five equal
equal	two equal
variable	while in else
colon	colon
endText	in eight else

Table 7.2: The recognition of the sample 2/3.mp3.

7.3 Using a custom grammar

After that, one way to increase the precision is to use a custom grammar. This allows us to restrict the possible sentences uttered to a smaller set of valid sentences, reducing possible misrecognition. We test two grammars: the full python grammar (taken from the python documentation, as described in Appendix B), and a simplified grammar. The simplified grammar use smaller lines and removes lots of possibilities, but greatly reduces the number of possibilities of sentences (and thus of errors). The simplified grammar forces each line to do one thing, like an assignation, comparison or a control statement. In a certain way it is lower level, and greatly reduces the risk of mixup. It is available in Appendix B

As we can see in table 7.3, using a grammar can improve the sentences uttered. There is no more noise or small parasite words, and some sentences are correctly recognized. But we can see a problem arising. When the recognizer starts to recognize a sentence not uttered by the user, it continues into that sentence, propagating the error until it finishes the sentence. This can be seen in table 7.3 with the plus eight in the simplified grammar, and even more in the full grammar and the "in eight in eight plus two divide eight multiply two endLine": had it not recognized endLine, it could have continued a long time into this sentence.

We can also see that the simplified grammar is way better for the recognition; the problem is that this grammar is a small subset, and is a workaround instead of a solution. But we can conclude that a smaller grammar yields better results.

7.4 Using a custom acoustic model

The acoustic model is the model that allows the recognition engine to correctly recognize and classify sounds. A full acoustic model is very long to make, requiring hundreds of hours of recording, weeks of adaptation and fine-tuning, and is a tedious process. Fortunately for us, there are already several high-quality acoustic models available on the Internet. In our case, one of those models come by default with PocketSphinx, and the other tested yielded the same results as the default one. But one thing we can do is to adapt an existing acoustic model using our data to make it better with our use case.

This process of adapting an acoustic model has a few steps. First, we must create a transcript of multiple sentences that could be uttered by the user in a .transcription file. After that, we must record the said sentences, and save them in a 16Khz mono 16bit .wav file, and name them according to the specification of the .transcript file. Once this is done, we can convert the .wav file into .mfc files using sphinx_fe, which is part of the sphinx toolkit. After that, the program pocketsphinx_mdef_convert, bw and mk_s2sendump (all from the Sphinx toolkit)

Text uttered	no grammar	simplified grammar	full grammar
variable	one eight elif	variable	variable
equal	yield	equal	equal
five	while five eight if	five	five
endLine	in in eight	plus eight endLine	in eight in eight
if	if yield two one	raise	plus two
variable	while yield	variable	divide eight
colon	colon in	endLine else	multiply two
endLine	in nine two	colon endLine	endLine
while	while in	elif	while
variable	nine eight	variable	variable
colon	else colon	colon	colon
endLine	in two nine	endLine	endLine
return	three def in	continue	yield two
variable	comma yield	endLine	in variable
endText	two in two else	continue	endText

Table 7.3: The recognition of the sample 4/1.mp3 using grammars

allow to process the .mfc files and include them into the acoustic model. At the bw stage some errors for the audio files can appear, such as

```
ERROR: "backward.c", line 421: Failed to align audio to transcript:
      final state of the search is not reached
ERROR: "baum_welch.c", line 324: arctic_0017 ignored
```

This happens where the recognizer cannot link the transcription to the audio file. If this happens for a few files these errors can be ignored, but if it happens for more files, it is a good idea to re-record those files and to restart the process (which can take some time).

After all that is done, we can use the adapted acoustic model.

After testing, we can see that it has a small improvement, mainly on the words that are not in the English language (like endLine), which can be logical since the basic acoustic model was not trained with those words. But for the rest, the recognition is still not the best: a bit better, but still not enough to be usable. And when using a grammar, it makes no difference.

One example of the improvement is shown in table 7.4.

Text uttered	default acoustic model	custom acoustic model
variable	for five eight	for five eight
equal	elif equal two	elif equal
variable	while eight colon if	variable in
multiply	three multiply	three multiply
five	in five	if five
endLine	two endLine	two endLine
break	in for	in for
endLine	eight eight	endLine
return	nine three two in	nine three two
variable	while eight	variable
endLine	while endText	while endLine

Table 7.4: recognition of the sample 3/2.mp3 using a custom acoustic model

7.5 Command line arguments

PocketSphinx has a lot of command-line arguments. Most of them are used to finely control the engine and do not impact much the quality of the recognition, but some of them can make an impact on the recognition. They are described in Appendix A, and most of them control the beamwidth (for the beam heuristic, with a beam width close to 1 discarding more results), the keyword spotting (with the control of the threshold or the delay), among other things. They all are default values that are supposed to be the best for most uses. So we tried to change them to see what changes. After testing multiple configurations of command-line arguments, we can realize that the arguments are very well-calibrated, and modifying them does not improve the quality of the recognition. Some make the recognition a little slower or faster, such as putting the different beam arguments to 0 slowing the recognition by about a quarter of a second (compared to the default beam value), which is enough to notice when comparing side by side, but without altering the results. Putting the beam arguments at 0.001 will lower the quality of the recognition (as expected), with the recognition looking more random than anything. The other arguments are in the same spot: They either lower the quality of the recognition or do not increase it.

This proves that the default arguments are well calibrated since they look like they are Pareto optimal. There is no way to increase quality, and increasing the speed is at the cost of a lot of quality.

7.6 Is PocketSphinx usable for coding with the voice ?

As we have seen in this chapter, the voice recognition provided by PocketSphinx is correct, but not sufficiently good. It works well for a series of commands with little branchings, such as "save file" or "open last project", but when the possibilities are increasing, there is too much confusion, even with a simplified grammar, it has a bad recognition quality.

This makes PocketSphinx at the moment unusable to dictate code. Code recognition is too complex, if the recognizer mixes up half (or even a quarter) of the words and the users have to use a keyboard to correct it, it becomes more of a burden than an actual help. This is too bad, as the solution implemented would be working correctly if the recognition was better.

Chapter 8

Conclusion

In this work, we have seen what are the possible alternatives for voice recognition in the browser, how the algorithms works, and we have explored in detail the functioning of pocketSphinx and if we could use it.

After research and benchmarking, we have found that although PocketSphinx is adequate for small projects with a relatively small grammar, for a bigger project with lots of branching and risk of mixup, this is not a suitable solution. Thus PocketSphinx is not fit for code recognition. The solution implemented would work without problem if the PocketSphinx recognition was more accurate.

Luckily, as the architecture of the project was created with modularity in mind, we will be able to swap the recognition engine with another better-performing engine. The problem is that, at the moment, there are no other obvious choices: there are not many projects for voice recognition in the open-source community. Kaldi is an interesting option, but its ecosystem of tools is not sufficiently mature, so it will require quite a lot of work to make it work: a WebAssembly build is mentioned in some forums, but is no more available (and apparently was not working flawlessly), and building it by hand failed. So the next step would be to implement a server communication with Kaldi and to check the quality of this voice recognition engine. This is doable with the Gstream extension but requires rebuilding the project and creating the javascript communication layer.

One other road would be to use non-open source services. As we have seen, most of the available server-based solutions are not fit for this use, but Dragon could be used. Even if it is not open source and costly, it is the biggest voice recognition engine available and used by many projects. But this would require works to implement, without even talking about the legal and monetary implications of using Dragon for the whole Ingenious website. The projects made with Dragon are usually projects that suppose the user has Dragon installed on his computer (and therefore no licensing issue), but to host it on a server would require a different license, and good security to avoid API spoofing (using the recognition without

being an Ingenuous user).

At the time of this writing, speech recognition is still growing. Large companies are leading the charge with enormous datasets usually taken (and labeled) without consent, and using heavy interpretation to transcribe text. But the open-source community is lagging, in part because of the lack of datasets, and even though some projects are arising to correct that, such as common voice by Mozilla [40], there is still work to do before open-source projects can be used instead of the big names. But those projects are growing and we can hope to have those projects gain more and more traction. Once they will be as good as the closed source, seeing that the concern for privacy is growing in the public opinion, we can hope they will force companies to respect more the privacy of their users.

Appendix A

Command line arguments of PocketSphinx

There is a multitude of command line arguments that can be passed to pocketsphinx to change its behaviour. Other than the arguments to specify acoustic models, dictionary, or logging level, some arguments can improve the quality of the recognition. Those arguments are [41] [42]

- `agc` : determine the automatic gain control, meaning this will influence the automatic correction of the audio volume level
- `agctresh` : set the threshold for automatic gain control
- `beam` : set the beam width applied to every frame in the Viterbi search
- `ds` : set the downsampling ratio for the frame GMM computation
- `kws_delay` : set the delay to wait for best detection score
- `kws_threshold` : set the threshold for recognition; a lower `kws_threshold` will trigger recognizing faster, meaning that background noise could be interfering, but a higher `kws_threshold` will maybe skip some sentences it should have recognized.
- `lpbeam` and `lponlybeam` : set the beam width that is applied to the last phonem in words. `lponlybeam` set it for single phonem words
- `pbeam` : set the beam width applied to the phonem transition
- `pl_window` : set the phoneme lookahead window size, used in the phonetic fast match algorithm

- wbeam : set the beam width applied to word exits
- topn : set the maximum number of top gaussians to use during the scoring process.
- maxwpmf : set the maximum number of distinct word exits at each frame
- maxhmmpf : set the maximum number of active HMMs to maintain at each frame

Using those arguments, we can have a lot of different configurations that will have different recognition quality.

Appendix B

Grammars used

B.1 The full python grammar in JSGF form

The full python grammar is quite big and quite long (about 140 lines): it was converted from the official python grammar using a tool I wrote in python. For a better reading experience, you can use go to the GitHub's project to download the file [39]

```
#JSGF V1.0;
grammar base;
public <single_input> = endLine | <simple_stmt> | <compound_stmt>
    ↪ endLine ;
<file_input> = ( endLine | <stmt> ) * ;
<eval_input> = <testlist> endLine * ;
<decorator> = at <dotted_name> [ openParenthesis [ <arglist> ]
    ↪ closeParenthesis ] endLine ;
<decorators> = <decorator> + ;
<decorated> = <decorators> ( <classdef> | <funcdef> | <
    ↪ async_funcdef> ) ;
<async_funcdef> = async <funcdef> ;
<funcdef> = def variable <parameters> [ arrow <test> ] colon [
    ↪ type_comment ] <func_body_suite> ;
<parameters> = openParenthesis [ <typedargslist> ] closeParenthesis
    ↪ ;
<typedargslist> = (
    ( <tfpdef> [ equal <test> ] ( comma [ type_comment ] <tfpdef> [
        ↪ equal <test> ] ) * comma [ type_comment ] divide [ comma
        ↪ [ [ type_comment ] <tfpdef> [ equal <test> ] (
```

```

comma [ type_comment ] <tfpdef> [ equal <test> ] ) * (
    ↪ type_comment | [ comma [ type_comment ] [
multiply [ <tfpdef> ] ( comma [ type_comment ] <tfpdef> [ equal
    ↪ <test> ] ) * ( type_comment | [ comma [ type_comment ] [
    ↪ pow <tfpdef> [ comma ] [ type_comment ] ] ] )
| pow <tfpdef> [ comma ] [ type_comment ] ] ] )
| multiply [ <tfpdef> ] ( comma [ type_comment ] <tfpdef> [
    ↪ equal <test> ] ) * ( type_comment | [ comma [
    ↪ type_comment ] [ pow <tfpdef> [ comma ] [ type_comment ]
    ↪ ] ] )
| pow <tfpdef> [ comma ] [ type_comment ] ] ] )
| ( <tfpdef> [ equal <test> ] ( comma [ type_comment ] <tfpdef>
    ↪ [ equal <test> ] ) * ( type_comment | [ comma [
    ↪ type_comment ] [
multiply [ <tfpdef> ] ( comma [ type_comment ] <tfpdef> [ equal
    ↪ <test> ] ) * ( type_comment | [ comma [ type_comment ] [
    ↪ pow <tfpdef> [ comma ] [ type_comment ] ] ] )
| pow <tfpdef> [ comma ] [ type_comment ] ] ] )
| multiply [ <tfpdef> ] ( comma [ type_comment ] <tfpdef> [
    ↪ equal <test> ] ) * ( type_comment | [ comma [
    ↪ type_comment ] [ pow <tfpdef> [ comma ] [ type_comment ]
    ↪ ] ] )
| pow <tfpdef> [ comma ] [ type_comment ] )
) ;
<tfpdef> = variable [ colon <test> ] ;
<varargslst> = <vfpdef> [ equal <test> ] ( comma <vfpdef> [ equal
    ↪ <test> ] ) * comma divide [ comma [ ( <vfpdef> [ equal <test>
    ↪ ] ( comma <vfpdef> [ equal <test> ] ) * [ comma [
multiply [ <vfpdef> ] ( comma <vfpdef> [ equal <test> ] ) * [
    ↪ comma [ pow <vfpdef> [ comma ] ] ]
| pow <vfpdef> [ comma ] ] ]
| multiply [ <vfpdef> ] ( comma <vfpdef> [ equal <test> ] ) * [
    ↪ comma [ pow <vfpdef> [ comma ] ] ]
| pow <vfpdef> [ comma ] ) ] ] | ( <vfpdef> [ equal <test> ] (
    ↪ comma <vfpdef> [ equal <test> ] ) * [ comma [
multiply [ <vfpdef> ] ( comma <vfpdef> [ equal <test> ] ) * [
    ↪ comma [ pow <vfpdef> [ comma ] ] ]
| pow <vfpdef> [ comma ] ] ]
| multiply [ <vfpdef> ] ( comma <vfpdef> [ equal <test> ] ) * [
    ↪ comma [ pow <vfpdef> [ comma ] ] ]

```

```

    | pow <vfpdef> [ comma ]
) ;
<vfpdef> = variable ;
<stmt> = <simple_stmt> | <compound_stmt> ;
<simple_stmt> = <small_stmt> ( semicolon <small_stmt> ) * [
    ↪ semicolon ] endLine ;
<small_stmt> = ( <expr_stmt> | <del_stmt> | <pass_stmt> | <
    ↪ flow_stmt> |
    <import_stmt> | <global_stmt> | <nonlocal_stmt> | <assert_stmt>
    ↪ ) ;
<expr_stmt> = <testlist_star_expr> ( <annassign> | <augassign> ( <
    ↪ yield_expr> | <testlist> ) |
    [ ( equal ( <yield_expr> | <testlist_star_expr> ) ) + [
    ↪ type_comment ] ] ) ;
<annassign> = colon <test> [ equal ( <yield_expr> | <
    ↪ testlist_star_expr> ) ] ;
<testlist_star_expr> = ( <test> | <star_expr> ) ( comma ( <test> |
    ↪ <star_expr> ) ) * [ comma ] ;
<augassign> = ( equalPlus | equalMinus | equalMultiply | equalAt |
    ↪ equalDivide | equalModulo | equalBinaryAnd | equalBinaryOr |
    ↪ equalBinaryXor |
    equalLeftShift | equalRightShift | equalPow |
    ↪ equalDivideEuclidian ) ;
<del_stmt> = del <exprlist> ;
<pass_stmt> = pass ;
<flow_stmt> = <break_stmt> | <continue_stmt> | <return_stmt> | <
    ↪ raise_stmt> | <yield_stmt> ;
<break_stmt> = break ;
<continue_stmt> = continue ;
<return_stmt> = return [ <testlist_star_expr> ] ;
<yield_stmt> = <yield_expr> ;
<raise_stmt> = raise [ <test> [ from <test> ] ] ;
<import_stmt> = <import_name> | <import_from> ;
<import_name> = import <dotted_as_names> ;
<import_from> = ( from ( ( dot | threeDot ) * <dotted_name> | ( dot
    ↪ | threeDot ) + )
    import ( multiply | openParenthesis <import_as_names>
    ↪ closeParenthesis | <import_as_names> ) ) ;
<import_as_name> = variable [ as variable ] ;
<dotted_as_name> = <dotted_name> [ as variable ] ;

```

```

<import_as_names> = <import_as_name> ( comma <import_as_name> ) * [
    ↪ comma ] ;
<dotted_as_names> = <dotted_as_name> ( comma <dotted_as_name> ) * ;
<dotted_name> = variable ( dot variable ) * ;
<global_stmt> = global variable ( comma variable ) * ;
<nonlocal_stmt> = nonlocal variable ( comma variable ) * ;
<assert_stmt> = assert <test> [ comma <test> ] ;
<compound_stmt> = <if_stmt> | <while_stmt> | <for_stmt> | <try_stmt>
    ↪ > | <with_stmt> | <funcdef> | <classdef> | <decorated> | <
    ↪ async_stmt> ;
<async_stmt> = async ( <funcdef> | <with_stmt> | <for_stmt> ) ;
<if_stmt> = if <namedexpr_test> colon <suite> ( elif <
    ↪ namedexpr_test> colon <suite> ) * [ else colon <suite> ] ;
<while_stmt> = while <namedexpr_test> colon <suite> [ else colon <
    ↪ suite> ] ;
<for_stmt> = for <exprlist> in <testlist> colon [ type_comment ] <
    ↪ suite> [ else colon <suite> ] ;
<try_stmt> = ( try colon <suite>
    ( ( <except_clause> colon <suite> ) +
    [ else colon <suite> ]
    [ finally colon <suite> ] |
    finally colon <suite> ) ) ;
<with_stmt> = with <with_item> ( comma <with_item> ) * colon [
    ↪ type_comment ] <suite> ;
<with_item> = <test> [ as <expr> ] ;
<except_clause> = except [ <test> [ as variable ] ] ;
<suite> = <simple_stmt> | endLine indent <stmt> + dedent ;
<namedexpr_test> = <test> [ equalColon <test> ] ;
<test> = <or_test> [ if <or_test> else <test> ] | <lambdef> ;
<test_nocond> = <or_test> | <lambdef_nocond> ;
<lambdef> = lambda [ <varargslist> ] colon <test> ;
<lambdef_nocond> = lambda [ <varargslist> ] colon <test_nocond> ;
<or_test> = <and_test> ( or <and_test> ) * ;
<and_test> = <not_test> ( and <not_test> ) * ;
<not_test> = not <not_test> | <comparison> ;
<comparison> = <expr> ( <comp_op> <expr> ) * ;
<comp_op> = lowerThan | greaterThan | doubleEqual |
    ↪ greaterOrEqualThan | lowerOrEqualThan | airplane | different
    ↪ | i binaryOr <not'> i binaryOr <is> binaryOr <is'> not ;
<star_expr> = multiply <expr> ;

```

```

<expr> = <xor_expr> ( binaryOr <xor_expr> ) * ;
<xor_expr> = <and_expr> ( binaryXor <and_expr> ) * ;
<and_expr> = <shift_expr> ( binaryAnd <shift_expr> ) * ;
<shift_expr> = <arith_expr> ( ( leftShift | rightShift ) <
    ↪ arith_expr> ) * ;
<arith_expr> = <term> ( ( plus | minus ) <term> ) * ;
<term> = <factor> ( ( multiply | at | divide | modulo |
    ↪ divideEuclidian ) <factor> ) * ;
<factor> = ( plus | minus | tilde ) <factor> | <power> ;
<power> = <atom_expr> [ pow <factor> ] ;
<atom_expr> = [ await ] <atom> <trailer> * ;
<atom> = ( openParenthesis [ <yield_expr> | <testlist_comp> ]
    ↪ closeParenthesis |
    openBrackets [ <testlist_comp> ] closeBracketss |
    openCurly [ <dictorsetmaker> ] closeCurly |
    variable | number | string + | threeDot | None | True | False )
    ↪ ;
<testlist_comp> = ( <namedexpr_test> | <star_expr> ) ( <comp_for> |
    ↪ ( comma ( <namedexpr_test> | <star_expr> ) ) * [ comma ] ) ;
<trailer> = openParenthesis [ <arglist> ] closeParenthesis |
    ↪ openBrackets <subscriptlist> closeBracketss | dot variable ;
<subscriptlist> = <subscript> ( comma <subscript> ) * [ comma ] ;
<subscript> = <test> | [ <test> ] colon [ <test> ] [ <sliceop> ] ;
<sliceop> = colon [ <test> ] ;
<exprlist> = ( <expr> | <star_expr> ) ( comma ( <expr> | <star_expr>
    ↪ > ) ) * [ comma ] ;
<testlist> = <test> ( comma <test> ) * [ comma ] ;
<dictorsetmaker> = ( ( ( <test> colon <test> | pow <expr> )
    ( <comp_for> | ( comma ( <test> colon <test> | pow <expr> ) ) *
        ↪ [ comma ] ) ) |
    ( ( <test> | <star_expr> )
    ( <comp_for> | ( comma ( <test> | <star_expr> ) ) * [ comma ] )
        ↪ ) ) ;
<classdef> = class variable [ openParenthesis [ <arglist> ]
    ↪ closeParenthesis ] colon <suite> ;
<arglist> = <argument> ( comma <argument> ) * [ comma ] ;
<argument> = ( <test> [ <comp_for> ] |
    <test> equalColon <test> |
    <test> equal <test> |
    pow <test> |

```

```

    multiply <test> ) ;
<comp_iter> = <comp_for> | <comp_if> ;
<sync_comp_for> = for <exprlist> in <or_test> [ <comp_iter> ] ;
<comp_for> = [ async ] <sync_comp_for> ;
<comp_if> = if <test_nocond> [ <comp_iter> ] ;
<encoding_decl> = variable ;
<yield_expr> = yield [ <yield_arg> ] ;
<yield_arg> = from <test> | <testlist_star_expr> ;
<func_body_suite> = <simple_stmt> | endLine [ type_comment endLine
    ↪ ] indent <stmt> + dedent ;
<func_type_input> = <func_type> endLine * ;
<func_type> = openParenthesis [ <typelist> ] closeParenthesis arrow
    ↪ <test> ;
<typelist> = ( <test> ( comma <test> ) * [ comma
    [ multiply [ <test> ] ( comma <test> ) * [ comma pow <test> ] |
    ↪ pow <test> ] ]
    | multiply [ <test> ] ( comma <test> ) * [ comma pow <test> ] |
    ↪ pow <test> ) ;

```

B.2 The full python grammar in FSG form

The full python grammar in fsg form is way too big to be shown here; it has about 75.000 nodes and 180.000 transitions (and lines) ! A text file containing the fsg form is available on the github. Here is the begining of this file, to show in which format a data file store FSG (this format can be easily parsed into javascript code, or loaded directly).

```

FSG_BEGIN <base.single_input>
NUM_STATES 73062
START_STATE 0
FINAL_STATE 72052
TRANSITION 0 13093 1.000000 at
TRANSITION 0 50417 1.000000 for
TRANSITION 0 72092 1.000000 import
TRANSITION 0 22548 1.000000 class
TRANSITION 0 62944 1.000000 if
TRANSITION 0 13066 1.000000 at
TRANSITION 0 72052 1.000000 endLine
TRANSITION 0 72053 1.000000 assert
TRANSITION 0 72106 1.000000 yield

```

```

TRANSITION 0 72055 1.000000 pass
TRANSITION 0 72055 1.000000 break
TRANSITION 0 72055 1.000000 continue
TRANSITION 0 72503 1.000000 test
TRANSITION 0 72503 1.000000 multiply
TRANSITION 0 72057 1.000000 nonlocal
TRANSITION 0 72061 1.000000 global
TRANSITION 0 32209 1.000000 try
TRANSITION 0 1 1.000000 async
TRANSITION 0 72112 1.000000 raise

```

B.3 The simplified python grammar in JSGF form

```

#JSGF V1.0;

grammar base;

public <allLine> = ([<addTab> | <removeTab> ])* <middleLine> <endingSentence>*;

<middleLine> = <definitionLine> | <definitionLine> |
    <functionLine> | <returnLine> | <conditionLine> |
    <assignationLine> | <whileLine> | <otherLine>;

<definitionLine> = def <functionName> openParenthesis
    [<variable> comma]* closeParenthesis colon;
<functionLine> = <functionCall>;
<whileLine> = while variable colon;
<returnLine> = (return | yield) variable;
<conditionLine> = (if variable colon) | (elif variable colon);
<assignationLine> = variable equal (<operation>|<functionCall>);
<otherLine> = break | continue | (try colon) |
    (except [<variable>] colon) | (finally colon) |
    (raise <variable>) | (else colon) | pass;

<functionCall> = <functionName> openParenthesis
    [<variable> [ (comma) <variable>]*] closeParenthesis;
<operation> = (<variable> | <digit>)
    [(plus|minus|divide|multiply|modulo|equal) (<variable>|<digit>)];

<number> = [<digit>]+;

```

```
<digit> = ( one | two | three | four | five | six | seven | eight | nine);  
<variable> = variable;  
<functionName> = function;  
<className> = class;  
<endingSentence> = endLine | endText;
```

Appendix C

International Phonetic Alphabet

In the figure C.1 you can see the correspondence between the CMU pronouncing dictionary and the International Phonetic Alphabet.

CMU form	IPA form	CMU form	IPA form	CMU form	IPA form
B	b	P	p	AH	ə
CH	tʃ	R	r	AO	ɔ
D	d	S	s	EH	ɛ
DH	ð	SH	ʃ	ER	ə
F	f	T	t	IH	ɪ
G	g	TH	θ	IY	i
HH	h	V	v	UH	ʊ
JH	dʒ	W	w	UW	u
K	k	Y	j	AW	aʊ
L	l	Z	z	AY	aɪ
M	m	ZH	ʒ	EY	eɪ
N	n	AA	ɑ	OW	oʊ
NG	ŋ	AE	æ	OY	ɔɪ

Figure C.1: the CMU alphabet and its IPA counterpart

Bibliography

- [1] “Electron apps.” [Online]. Available: <https://electronjs.org/apps>
- [2] “Web Speech API,” Sep. 2019. [Online]. Available: <https://wicg.github.io/speech-api/>
- [3] “Statcounter global stats.” [Online]. Available: <https://gs.statcounter.com/>
- [4] “Desktop internet browser market share 2015-2019.” [Online]. Available: <https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/>
- [5] “Web Speech API.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API
- [6] “Microsoft Edge Insider.” [Online]. Available: <https://www.microsoftedgeinsider.com/en-us/>
- [7] “Download Microsoft Edge, based on Chromium.” [Online]. Available: <https://support.microsoft.com/en-us/help/4501095/download-microsoft-edge-based-on-chromium>
- [8] “Unable to use microphone (affects Duolingo, Google Translate, and other sites) · Issue #3725 · brave/brave-browser.” [Online]. Available: <https://github.com/brave/brave-browser/issues/3725>
- [9] “Secure, Fast & Private Web Browser with Adblocker.” [Online]. Available: <https://brave.com/>
- [10] “cmusphinx/pocketsphinx,” Nov. 2019, original-date: 2014-04-07T10:28:32Z. [Online]. Available: <https://github.com/cmusphinx/pocketsphinx>
- [11] “WebAssembly.” [Online]. Available: <https://webassembly.org/>
- [12] “Pocketsphinx.js - Speech Recognition in JavaScript and WebAssembly.” [Online]. Available: <https://syl22-00.github.io/pocketsphinx.js/>

- [13] S. Chevalier, “syl22-00/pocketsphinx.js,” Nov. 2019, original-date: 2013-06-01T12:47:39Z. [Online]. Available: <https://github.com/syl22-00/pocketsphinx.js>
- [14] “Dragon Speech Recognition - Get More Done by Voice.” [Online]. Available: <https://www.nuance.com/dragon.html>
- [15] “Kaldi: About the Kaldi project.” [Online]. Available: <https://kaldi-asr.org/doc/about.html>
- [16] N. Shmyrev, “Update on CMUSphinx Project.” [Online]. Available: <http://cmusphinx.github.io/2019/10/update/>
- [17] J. G. David Forney, “The viterbi algorithm,” 1973.
- [18] “Beam search,” Sep. 2019, page Version ID: 915535252. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Beam_search&oldid=915535252
- [19] J. Brownlee, “How to Implement a Beam Search Decoder for Natural Language Processing,” Jan. 2018. [Online]. Available: <https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/>
- [20] D. Povey, D. Kanevsky, B. Kingsbury, B. Ramabhadran, G. Saon, and K. Visweswariah, “Boosted MMI for model and feature-space discriminative training,” in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, March 2008, pp. 4057–4060.
- [21] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, “The Kaldi Speech Recognition Toolkit,” in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, Dec. 2011, iEEE Catalog No.: CFP11SRW-USB.
- [22] “Using python to code by voice.” [Online]. Available: <https://www.youtube.com/watch?v=8SkdfdXWYal>
- [23] “Advanced Voice-Control, speech to code, program by voice, stop RSI.” [Online]. Available: <https://voicecode.io/>
- [24] “Vocola - A Voice Command Language.” [Online]. Available: <http://vocola.net/>
- [25] A. Hunt, “JSpeech Grammar Format,” Jun. 2000. [Online]. Available: <https://www.w3.org/TR/2000/NOTE-jsgf-20000605/>

- [26] “JSGFGrammar (Sphinx-4).” [Online]. Available: <https://www.gavo.t.u-tokyo.ac.jp/~kuenishi/java/sphinx4/edu/cmu/sphinx/jsapi/JSGFGrammar.html>
- [27] “Java Speech Grammar Format.” [Online]. Available: https://software.intel.com/sites/landingpage/realSense/camera-sdk/v1.1/documentation/html/doc_speech_specifying_commands_using_java.html
- [28] “10. Full Grammar specification — Python 3.8.0 documentation.” [Online]. Available: <https://docs.python.org/3/reference/grammar.html>
- [29] “ARPABET,” Oct. 2019, page Version ID: 922635615. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=ARPABET&oldid=922635615>
- [30] “CMU Pronouncing Dictionary,” Oct. 2019, page Version ID: 923225758. [Online]. Available: https://en.wikipedia.org/w/index.php?title=CMU_Pronouncing_Dictionary&oldid=923225758
- [31] “International Phonetic Alphabet,” Nov. 2019, page Version ID: 927381179. [Online]. Available: https://en.wikipedia.org/w/index.php?title=International_Phonetic_Alphabet&oldid=927381179
- [32] “The CMU Pronouncing Dictionary.” [Online]. Available: <http://www.speech.cs.cmu.edu/cgi-bin/cmudict#phones>
- [33] “Using Web Workers.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
- [34] “WebAssembly.” [Online]. Available: <https://webassembly.org/>
- [35] N. Shmyrev, “Adapting the default acoustic model.” [Online]. Available: <http://cmusphinx.github.io/wiki/tutorialadapt/>
- [36] J. F. js.foundation, “How to Create a Basic Plugin | jQuery Learning Center.” [Online]. Available: <https://learn.jquery.com/plugins/basic-plugin-creation/>
- [37] F. P. Miller, A. F. Vandome, and J. McBrewster, *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau-Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009.
- [38] N. Shmyrev, “CMUSphinx Tutorial For Developers.” [Online]. Available: <http://cmusphinx.github.io/wiki/tutorial/>
- [39] [Online]. Available: <https://github.com/gonzeD/IngiVoice>

- [40] [Online]. Available: <https://voice.mozilla.org/en>
- [41] “cmusphinx/pocketsphinx.” [Online]. Available: https://github.com/cmusphinx/pocketsphinx/blob/master/doc/pocketsphinx_continuous.1
- [42] N. Shmyrev, “CMUSphinx Open Source Speech Recognition.” [Online]. Available: <http://cmusphinx.github.io/wiki/pocketsphinxhandhelds/>

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl