



**School of Science and Engineering**

Capstone Design

EGR 4402-01 HC

Capstone Report

# **Using NLP and Supervised Learning to Learn Moroccan Tamazight Language**

Abdelmajid Essofi

Supervised by Dr. Violetta Cavalli-Sforza

Fall 2019

**School of Science and Engineering - AL AKHAWAYN UNIVERSITY**

## List of Figures

Figure 1: Steeple (Pestel) Analysis Diagram .....	6
Figure 2: Hypothesis Initialization.....	11
Figure 3: Specific Hypothesis Adjustment .....	12
Figure 4: General Hypotheses Adjustment.....	12
Figure 5: Tree Representation of a Version Space .....	15
Figure 6: Front-end Technologies Used.....	16
Figure 7: NodeJs for Bootstrap and Other Package Installations .....	16
Figure 8: Backend Technologies Used .....	17
Figure 9: User Interface, Initial View .....	17
Figure 10: Pattern Example with Dropdown List Content .....	18
Figure 11: Pattern Removal Process.....	18
Figure 12: Input Sample and Autocomplete with Substring Feature.....	19
Figure 13: Latin-Tifinagh Letter Conversion Scale .....	21
Figure 14: User Request for Number of Examples .....	21
Figure 15: 3 System-Generated Examples.....	22
Figure 16: Layout of the Word Addition Form.....	23
Figure 17: Word Submission Modal .....	24
Figure 18: Autocomplete Returning No Results on Search .....	24
Figure 19: Word Successfully Added to the Autocomplete List .....	25
Figure 20: Training Data Set Structure.....	27
Figure 21: Dictionary Structure of Common Nouns .....	28
Figure 22: Initial Structure of Training Hypotheses .....	30
Figure 23: Hypothesis Update on Negative Example.....	32
Figure 24: Converged Hypothesis to Final Answer .....	33
Figure 25: Combining Specific Hypotheses for Sentence Generation.....	34
Figure 26: DFS Traversal for Sentence Generation.....	35
Figure 27: Recursive Tree of the Sentence-Generation Algorithm.....	36

# Table of Contents

1. Introduction .....	4
2. STEEPLE Analysis.....	6
3. Methodology.....	8
4. Version Space Algorithm .....	10
4.1. How It Works.....	10
4.2. General Process Definition .....	13
4.3. How It Ends .....	14
5. Implementation .....	16
5.1. User Interface .....	17
5.1.1. Choosing a Sentence Pattern.....	17
5.1.2. Feeding a Sample Sentence to the System.....	19
5.1.3. How Tifinagh Input is Guaranteed.....	20
5.1.4. System-Generated Examples .....	21
5.1.5. Updating the Dictionary.....	23
5.2. Backend Design.....	25
5.2.1. Training Set Structure .....	25
5.2.2. Dictionary Structure .....	27
5.2.3. Training Process.....	29
5.2.4. Sentence Generation Process .....	34
6. Final Remarks .....	38
6.1. Limitations and Challenges .....	38
6.2. Future Work.....	39
References .....	40

## 1. Introduction

Throughout the chapters of this report, I will be talking about my capstone project, from an initial description of the project, to the implementation process of the system. The object of this capstone is to build an intelligent system that uses Natural Language Processing (NLP) techniques to build grammatically and syntactically correct sentences in Tamazight language, initially following grammar rules taught to the system by a teacher. The system will learn through a process of supervised learning based on the feedback it receives from the teacher when it tries to use the rules it learned to propose sentences. This will allow the system in later steps to generate exercises with solutions to help teach the language to students.

The project is divided to two subparts, a machine learning backend, which is to be coded in Python, and a front-end user interface, developed in Angular 8, which is a JavaScript framework.

The development of this project requires a combination of automated processing, performed by the intelligent system, as well as a manual side, which consists of a data collection process performed by the developer. This manual process is to be achieved is by defining the dictionary of words, along with their characteristics, which will be used by the system to generate grammatically correct sentences. Grammatical correctness of a sentence is made by following the agreement that sentences have with each other with regards to their characteristics, such as: Gender, Number, Person, Tense, Aspect... Therefore, the goal of this system is to build an algorithm capable of using a data set consisting of sentences, labeled as correct or incorrect examples, and apply a learning algorithm on each sentence, in order to eventually develop a hypothesis that satisfies all the agreements that form a correct sentence. Because it is impractical and unrealistic to generate a ready-to-use data set that will provide the system with all the examples it will need to learn all the correct sentence structures from the first try, I have decided to develop a simple user interface through which the user, who is expected to be a knowledgeable teacher about the structure of Tamazight, will interact with the system. The user will be able to send the system a set of sentences

that will be labeled as correct or incorrect ones, which the system will then use to adjust its learnt hypotheses in order to generate more correct sentences as the process is repeated. As the system generated sentences, it is going to send them back to the user for feedback. The user will have the ability to send back a verdict to the system, where it will judge the generated sentence as a correct, or incorrect one. The process of learning and adjusting hypotheses is based on both the sentences generated by the user, as well as the ones generated by the system. The learning process will use, among other resources, individual words, extracted from a dictionary, which will cover a suitable subset of the language. I will also have to differentiate between different classes of words of the language (nouns, verbs, adjectives, pronouns, conjunctions...), which will help facilitate the composition of legal combinations of words to formulate correct sentences. Finally, I will need to be concerned with interactions between elements of the sentence.

## 2. STEEPLE Analysis

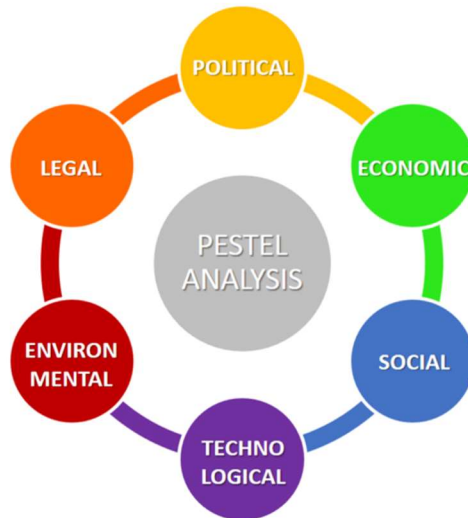


Figure 1: Steeple (Pestel) Analysis Diagram

### **Social Factor**

The use of the Tamazight language in Morocco by its native speakers has drastically decreased over the years. In contrast, during my time as a citizen of Morocco and university student, I have noticed an increasing interest in this language, both by Moroccans and by international students who specifically came to Morocco seeking an opportunity to learn Tamazight. Therefore, I believe that this project is a great opportunity to contribute to this societal interest and, in the long run, provide people with a platform that will guide them through their learning process and that they can use to at least acquire beginner knowledge of the language.

### **Technological Factor**

Technology has come a long way and, nowadays, the spotlight is often on Machine Learning. This project follows the trend, since it uses Natural Language Processing in order to help the system learn the language at hand. My project is a combination of NLP and Web Development, which joins Research and Development, as well as innovation in the technological field, into a single product.

### **Environmental Factor**

The development process of this project is highly environment friendly. If successfully developed, not only will it decrease the use of paper in a student's language learning process, but it will also not harm the environment in any major way during its development process. The one potential negative impact that this project can have on the environment comes from the computer's emissions (heat, noise...), which is a small price to pay for all the long-term advantages of this project.

### **Economic Factor**

Machine Learning, as mentioned above, is under the spotlight nowadays. To that extent, this project will be contributing to and taking advantage of a sector of the economy that is quickly developing and, in the future, may benefit from investment in further developing this technology. From a different perspective, the project's orientation is educational and therefore not directly intended to positively impact the economy, though there may be indirect benefits from its use through increasing communication with socially marginal groups.

### **Political Factor**

The idea of this project is to innovate in the use of technology used in teaching language. Neither the project nor the technology used in implementing it have substantial political implications, except to the extent that the language targeted is one that has recently received more attention in the political and educational sphere.

### **Legal Factor**

This project does not oppose nor does it potentially violate any laws, Moroccan or otherwise. My project aims to provide a learning opportunity for all and, if anything, contributes to reducing the discrimination that Tamazight, as well as its speakers, experience in their lifetime.

## **Ethical Factor**

The purpose of this project is purely educational. In addition to experimenting with new techniques for building language learning platforms, its goal is to help in teaching language to students in universities, schools, or organizations. If it was ever to be used outside of an academic setting, this project would be given for good use and at no charge to organizations who aim to use it in adherence to ethical standards in order to help student get a good grasp of Tamazight language.

## **3. Methodology**

When I talked to my supervisor about this project, she provided me with material on Natural Language Processing and its algorithms, Machine Learning algorithms, and tools that we can use to develop a working system that serves our purpose. The first book I was provided with is titled Speech and Language Processing (Jurafsky & Martin, 2014). Another book I was provided with is a foundational book in the field and is titled Machine Learning (Mitchell, 1997). These books give an in-depth insight about different techniques and algorithms, commonly used in the automated processing of spoken languages. Such algorithms and techniques include Version Spaces, Regular Expressions, Language Modeling, Classification algorithms, Semantic Role Labeling, etc... These tools are very commonly used when parsing sentences and grammar rules, and their power mainly lies within their flexibility across languages. Because these techniques deal with string parsing as an independent process from the language being handled, it allows us to perform smart processes and implement algorithms on any language, no matter how complex. Therefore, despite the fact that, although supervised learning of the Tamazight language has not been addressed before, these tools and algorithms can be customized and adapted to learn Tamazight, if fed with the right information.

The techniques mentioned by these books are mostly significant, and very useful in Natural Language Processing. However, when I searched through previous implementations and publications that were related to these



techniques mentioned above, I found that they were mainly used in classification models. Such models are used to give verdicts about a certain pattern, and classify them into categories. When given a pattern, the model would study it, along with previously seen patterns, and classify them in one of the predefined categories. The goal of my project was far beyond that: Given the previously seen patterns, along with their labeling as correct or incorrect sentences, it is to generate grammatically meaningful patterns and full sentences, and return them to the user as an output. For that reason, based on the articles that my supervisor gave me and advised me to read, and after a thorough discussion with my supervisor about the algorithm to use, we agreed that the Version Space algorithm is the best way to go about it. However, because Natural Language Processing is a developing field, most algorithms have not yet been implemented, and used enough to be embedded in programming language libraries. Therefore, I have decided to go for a Version Space algorithm to generate my sentences, however, Python has no library that implements this algorithm, and that is why I decided to understand the algorithm, and implement its logic from scratch.

The technologies I have decided to use are: For the front-end part of the project, it was developed using Angular 8, a JavaScript framework developed by Google, made to create easy-to-develop user interfaces with computer and mobile compatibility, For the backend, I have decided to go for the most widely used programming language in Machine Learning in our days, Python. This choice was made because of Python's compatibility with Natural Language Processing tools and algorithms, as well as the easy manipulation that Python offers with regards to lists, data sets, and strings.

## 4. Version Space Algorithm

A version space is a tree representation of the learning process of the system, it enables keeping track of the needed information found in the training data set, without actually remembering any of the examples from the set.

The representation of the version space can take any form that will guarantee the system understanding of the structure that the training data comes in. For the sake of simplicity, I will use a dummy example, then apply the version space to my project in the implementation part of this report.

Is it a good day to take the bus on our way to a restaurant? Or is it better to walk? A good way to make this decision would be to use the version space algorithm.

### 4.1. How It Works

The first things to be provided with are a training set containing data about previous days, along with the best option for the conditions of that example (i.e. whether it is a good idea to take the bus or not), along with a set of parameters that we can use to make a decision. The parameters we are going to use for this test are: The state of the weather (Rainy, Sunny, Cloudy), the temperature of that day (Cold, Warm, Hot), the hunger state (Hungry, Not Hungry), and how far the restaurant is (10 minute walk or more, less than a 10 minute walk).

Once we have all of these conditions, it is time to start training our model. In order to start with the training process, we have to initialize our hypotheses. Hypotheses are divided to two kinds: A general hypothesis, and a specific one. The general hypothesis must be initialized to the combination of parameters that covers all possible states of the day. That can be done by initializing every parameter in our hypothesis to require no specific value, we define such parameters by an interrogation mark '?'. Then we must initialize our specific hypothesis to represent the first positive example in our data set. We define a positive example as a day in which we can walk to the restaurant rather than take the bus.

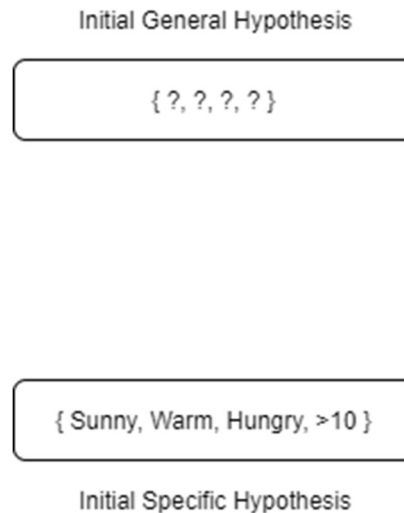


Figure 2: Hypothesis Initialization

The figure above represents the initial form of both the specific, as well as general hypotheses. The general hypothesis takes the form of a broad hypothesis, which accepts any possible input for any parameter as a correct one. The mostly specific hypothesis takes the form of the first positive hypothesis in our training set, which shows that the day is sunny, warm, that the person is hungry, and that the restaurant is more than a 10-minute walk away.

As we loop through the training set, we handle different examples differently. Positive examples are used to adjust specific hypotheses and generalize them, while negative examples are used to specialize the general hypotheses. For example, given the following positive example:

**{ Sunny, Cold, Hungry, > 10 }**, our hypotheses will be adjusted as follows:

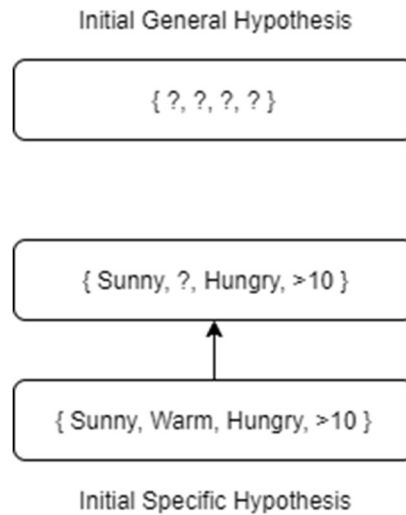


Figure 3: Specific Hypothesis Adjustment

Then, we adjust our general hypotheses by using negative examples from the dataset. For example, given the following:

**{ Rainy, Cloudy, Not Hungry, > 10 }**, our hypotheses will be adjusted as follows:

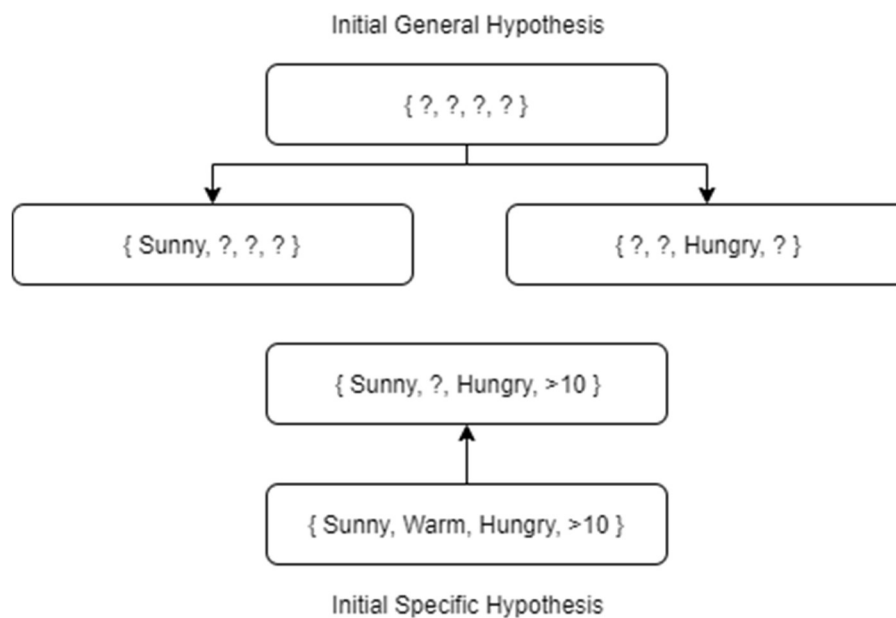


Figure 4: General Hypotheses Adjustment

For the negative examples, we don't know which parameter of the example made the example a wrong one. This is why, when handling a negative example, we keep track of all of our specific hypotheses, and search for disagreement between those and the negative example. If the specific hypothesis states that a parameter should support a different value than the one in the negative example, then we update our general hypothesis to include the value in the specific hypotheses as a separate hypothesis with a single changed value.

#### **4.2. General Process Definition**

The negative examples in the data set may be used mainly to handle and adjust general hypotheses, and positive ones are to be used to adjust specific hypotheses. However, this does not mean that the examples have no impact on the second type of hypotheses. Following the following process, we will learn that the specific hypotheses should be adjusted after every negative example as well, and the same goes for the general hypotheses, with regards to the positive examples.

Given our data set, along with our set of hypotheses, the algorithmic process goes as follows (DBD, 2019):

If the example in the data set is **Positive**:

- Change the specific hypotheses to exclude all disagreements they have with the example, and minimally generalize them to include the changes
  - Every specific example should only include minimal changes (For a hypothesis that requires  $n$  changes, create  $n$  separate hypotheses from it)
  - Every resulting specific example should still be more specific than all general examples
  - All specific examples should be equally specific
- Remove all general examples that disagree with the newly generated specific examples

If the example is **negative**:

- Change all general examples to be more specific versions of themselves in such a way that prevents them from matching with the negative example
  - Every new general hypothesis should only include minimal changes (For a hypothesis that requires  $n$  changes, create  $n$  separate hypotheses from it)
  - Every specialized general hypothesis should still be a general version of every specific hypothesis
  - All general hypotheses should be equally general to each other
- Exclude all specific models that agree with the negative example

#### 4.3. How It Ends

As the training process continues example after another, and the hypotheses being adjusted to the changes in every example, the algorithm ends if one of several scenarios occur (DBD, 2019):

- We use up all the examples in the training set
- We have a number of hypotheses equal to:
  - **Zero:** There is no way to describe the language from the training data set
  - **One:** We got our answer (The version space converges)
  - **Two or More:** The examples in the training data set include all descriptions of the language implicitly

If the Specific and General examples after the end of the training process are singletons, i.e. One hypothesis each, then it is one of two possible scenarios

- **They are identical:** This means that the version space converges to the right answer, which should be outputted to the user

- **They are different:** This is an indicator that the training data set examples were inconsistent with each other, we should output the result of this inconsistency and wait for better, more consistent, examples.
  - o This shows the main disadvantage of the version space algorithm, which is represented in the fact that noise and inconsistent data may cause the correct answer to be pruned away in the process.

The following figure is a tree representation, which summarizes what a version space looks like, when it ends, along with a comment describing every step of the process (DBD, 2019)

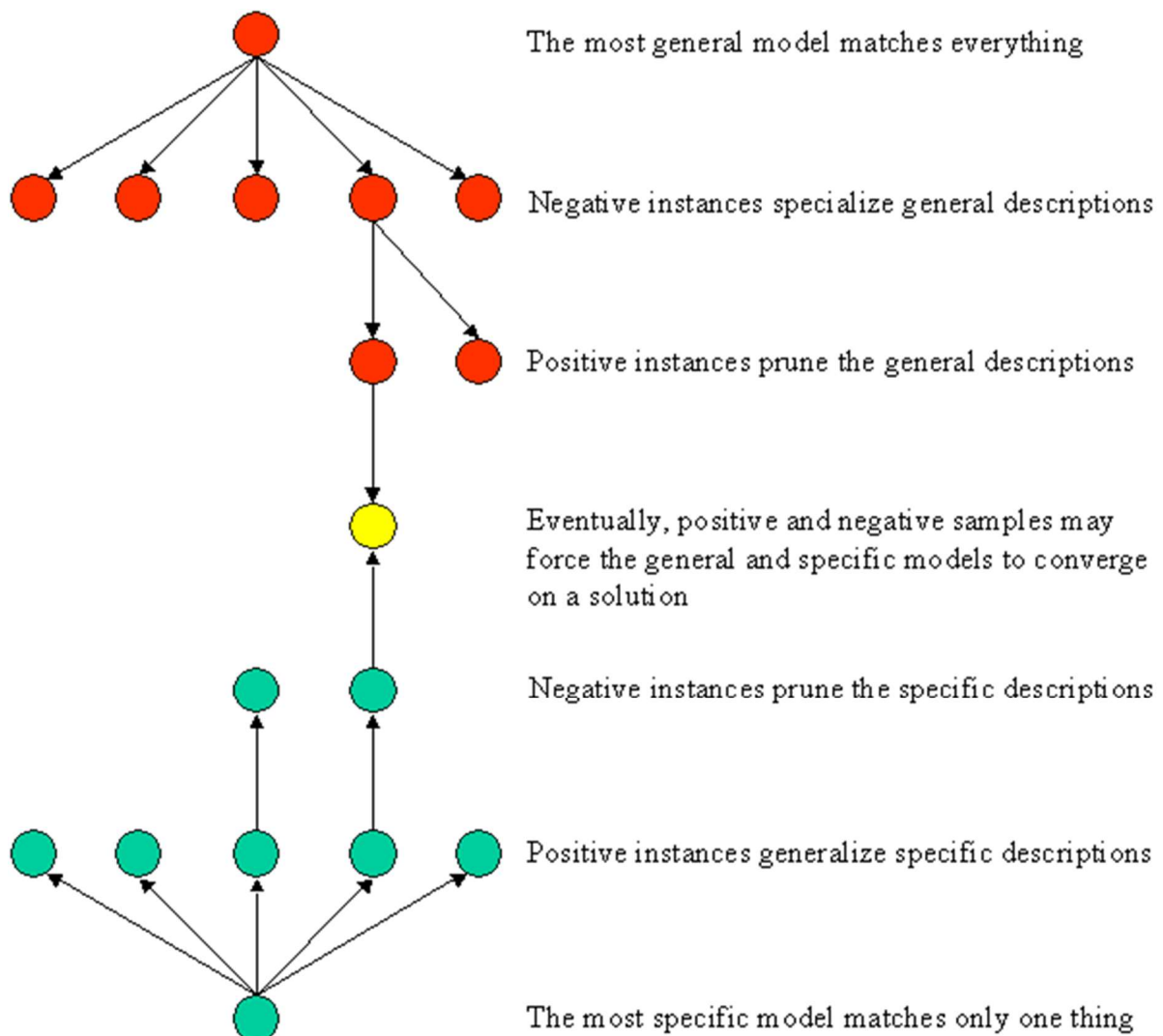


Figure 5: Tree Representation of a Version Space

## 5. Implementation

For the front-end implementation of this project, I had to divide my work into two subparts, a user interface for the user-system interaction, and a backend that implements the version space algorithm and supervised learning. JavaScript is the most widely used programming language for web development, so I decided to go for it for obvious reasons. However, JavaScript has an ocean of frameworks to choose from, each with their characteristics, advantages, and drawbacks. For the sake of comfort, simplicity, ease of manipulation, and previous experiences, I have decided to go with JavaScript's Angular 8 ("Angular", 2019) framework.



Figure 6: Front-end Technologies Used

I also needed some packages to go with my project, which I used to download all the libraries I needed such as Bootstrap (Otto, 2019), for the design and refinement of my web pages. For that, I counted on node (Foundation, 2019) packages to provide me with everything I needed.



Figure 7: NodeJs for Bootstrap and Other Package Installations

As for the backend logic, I have chosen to go for a programming language that will provide me with everything I will need to build my logic, without having to deal with any overhead or complexities. I also wanted a programming language that would provide me with Natural Language



Processing tools if need be. For that reason, I chose to go with Python ("Welcome to Python.org", 2019), but I still needed a version of Python that I can link with a web application, which is why, my backend is fully developed in Python's Flask ("Flask — Flask Documentation", 2019) framework.



Figure 8: Backend Technologies Used

## 5.1. User Interface

### 5.1.1. Choosing a Sentence Pattern

I had to keep some things in mind while designing and implementing the front-end design of this project. The user interface had to be as simple and straight forward as possible, one that the user can understand within 5 seconds of looking at a page. It also had to be very dynamic, and one that allows the user to control their patterns and sentences as they please, and that's by allowing insertion and deletion at any position of the sentence they want. Here's what the initial page of the project looks like upon the first visit:

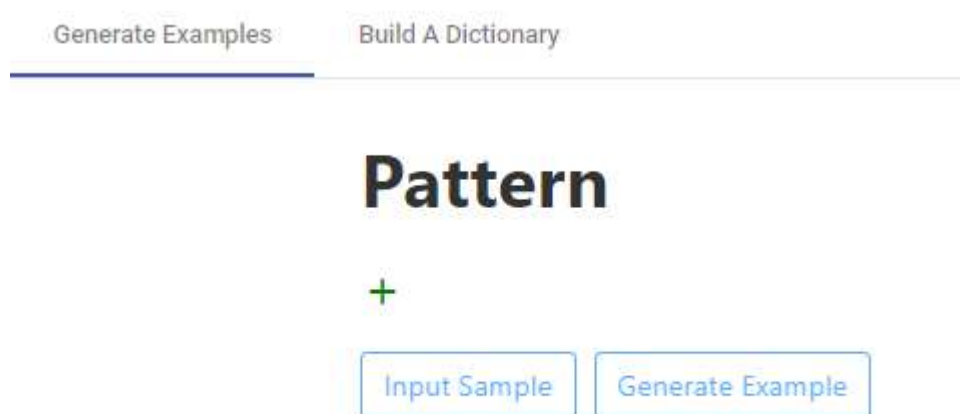


Figure 9: User Interface, Initial View

This design aims to attract the user's eye to the plus sign, which indicates the need to add more elements to the pattern we are trying to

build. The buttons allowing the user to input a sample sentence, or generate an example, are disabled until a pattern element is added.

As the user clicks on the '+' icon, more elements are added to the pattern, and the user has the freedom to select any one of the available parts of speech in the form of a dropdown list, with the continuous option to remove (By clicking '-') and add (By clicking '+') pattern elements anywhere at all times. The screenshot below shows an example of a possible pattern, followed by a screenshot showing the removal process of a pattern element from the list:

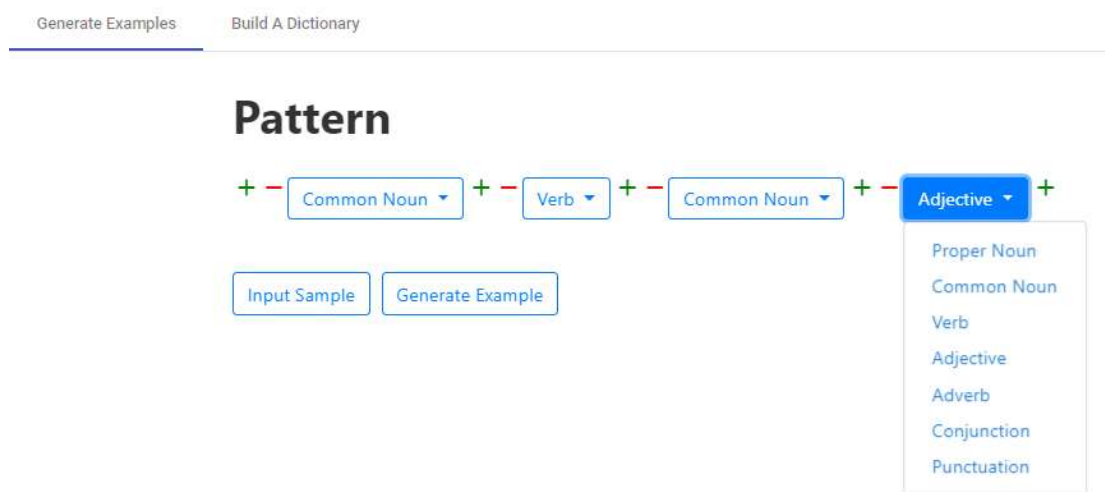


Figure 10: Pattern Example with Dropdown List Content

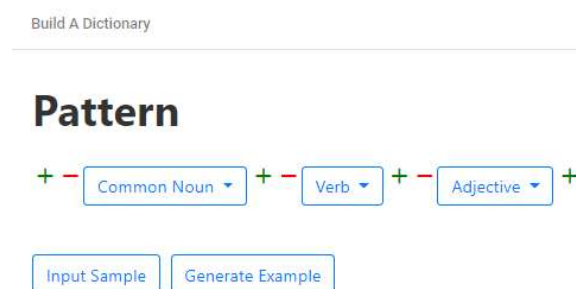
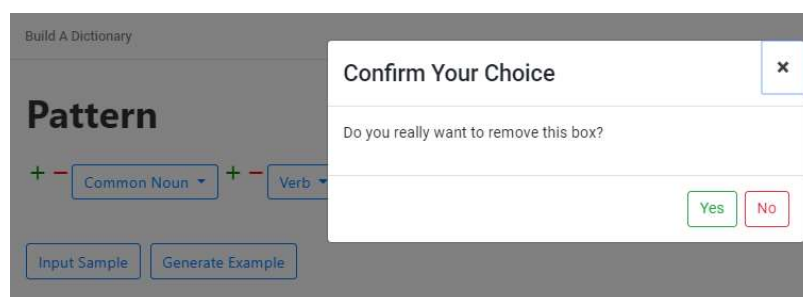


Figure 11: Pattern Removal Process

### 5.1.2. Feeding a Sample Sentence to the System

When the user has decided on the pattern they want to build their sentence on, they are allowed to input a sample sentence, which they can label as a correct or incorrect sentence, and send it to the system, which can then perform supervised training on that sentence, and add it to what they have learned about that pattern from previously fed sentences.

Once the Input Sample button is clicked, the user is provided with a number of input fields equal to the number of pattern elements they have above. Each input field is populated with words from the dictionary that match the part of speech corresponding to the selected input field. For example, if the third element of the pattern is a verb, the third input field is going to be populated with all the verbs from the dictionary. This is because the system implements an autocomplete feature, which works on a substring basis. Every time the user enters a string, they can only validate their input with one of the words from the list that contain the string entered so far by the user, which will all appear under the input field, after the autocomplete component has filtered all the options. Here is an example of what the input sample and the autocomplete feature look like on the interface:

Build A Dictionary

**Pattern**

+ - Verb +

**User Example**

ElO...

Submit Correct Example Submit Wrong Example Generate Example

**Pattern**

+ - Verb +

**User Example**

xq

+xq

+xq+ sample Submit Wrong Example Generate Example

Figure 12: Input Sample and Autocomplete with Substring Feature

As shown above, the user chose to enter a verb as a first element of our pattern, and when they clicked on the Input Sample button, they were provided with one input field, populated with all the verbs from the dictionary in all their forms, written in the Amazigh Tifinagh form. When the

user entered a set of characters, the autocomplete feature filtered the list of verbs in the dictionary to only leave the verbs on the list that had the user input as a substring in them. As shown above, “ⵗⵏ” is a substring of both “ⵜⵗⵏ” and “ⵜⵗⵏⵜ”, which are both different variations of the verb “ⵗ”, which is the verb “To Be” in Tamazight, written in Tifinagh alphabet.

### 5.1.3. How Tifinagh Input is Guaranteed

When I had a populated data set with Amazigh words, there was but one issue to handle. Tamazight has its own alphabet, which is called Tifinagh. The keyboard for Tifinagh is very accessible, and easily downloadable from the Keymann Desktop app. However, no one has a built-in Tifinagh keyboard showing on their laptop, or Tifinagh letters drawn on buttons, this creates a bit of a confusion, because the Latin to Tifinagh letter conversion is not always intuitive, and sometimes doesn’t even exist, such as ⵀ, ⵄ, ⵗ, ⵙ, which all have their equivalents in Arabic: ع غ خ ح respectively, but have no equivalent in Latin letters. To handle these characters, the downloadable keyboards just use a combination of characters that can be confusing to some users.

This is why I have decided to add an alternative for those who find the downloadable keyboards unpractical. I have taken advantage of the fact that Arabs have figured out a way to represent these special letters while chatting in Arabic using Latin letters. It is mostly the use of numbers. For the example above, the following combination of characters: ⵀ, ⵄ, ⵗ, ⵙ, is commonly translated by the following numbers: 3, 4, 5, 7. I trust that no one will find this conversion confusing, as it is used by all Arabs who use Latin letters while chatting in Arabic. The following table represents the conversion system that I have used, which allows the user of my system to type in Latin characters or numbers, and have them automatically replaced with their equivalent letter in Tifinagh as they type:

User Input	Tifinagh Equivalent	User Input	Tifinagh Equivalent	User Input	Tifinagh Equivalent	User Input	Tifinagh Equivalent
a	◌	g	ⵎ	h	ⵓ	c	ⵛ
b	ⵇ	t	ⵜ	w	ⵡ	4	ⵓ
d	ⵏ	m	ⵎ	n	ⵏ	5	ⵓ
z	⵵	j	ⵢ	l	ⵝ	7	ⵓ
f	ⵑ	q	ⵓ	i	ⵢ	3	ⵓ
k	ⵔ	r	ⵓ	s	ⵓ	y	ⵓ
T	ⵓ	D	ⵓ	Z	ⵓ	S	ⵓ

Figure 13: Latin-Tifinagh Letter Conversion Scale

#### 5.1.4. System-Generated Examples

When the user is done populating his dropdown list items with parts of speech of his choice to form a sentence pattern, his second option, after that of entering a sample sentence for the system to learn from, is to ask the system to generate a sentence, or a group of sentences, using what it has learnt so far about the selected pattern. For example, the user can select a pattern, consisting of a group of parts of speech, and ask the system to generate 3 sentences. The system should send this request to the backend server, informing it of the pattern requested, and the number of sentences required. The user should expect an output result in the following format:

Figure 14: User Request for Number of Examples

## Pattern

+ -  + -  + -  +

## System-Generated Example(s)

ⵍⵍⵉⵏⵏⵉⵢⵓⵏ ⵉⵎⵉⵏⵉⵢⵓⵏ ⵉⵎⵉⵏⵉⵢⵓⵏ	✓	✗
ⵉⵎⵉⵏⵉⵢⵓⵏ ⵉⵎⵉⵏⵉⵢⵓⵏ ⵉⵎⵉⵏⵉⵢⵓⵏ	✓	✗
ⵉⵎⵉⵏⵉⵢⵓⵏ ⵉⵎⵉⵏⵉⵢⵓⵏ ⵉⵎⵉⵏⵉⵢⵓⵏ	✓	✗

Figure 15: 3 System-Generated Examples

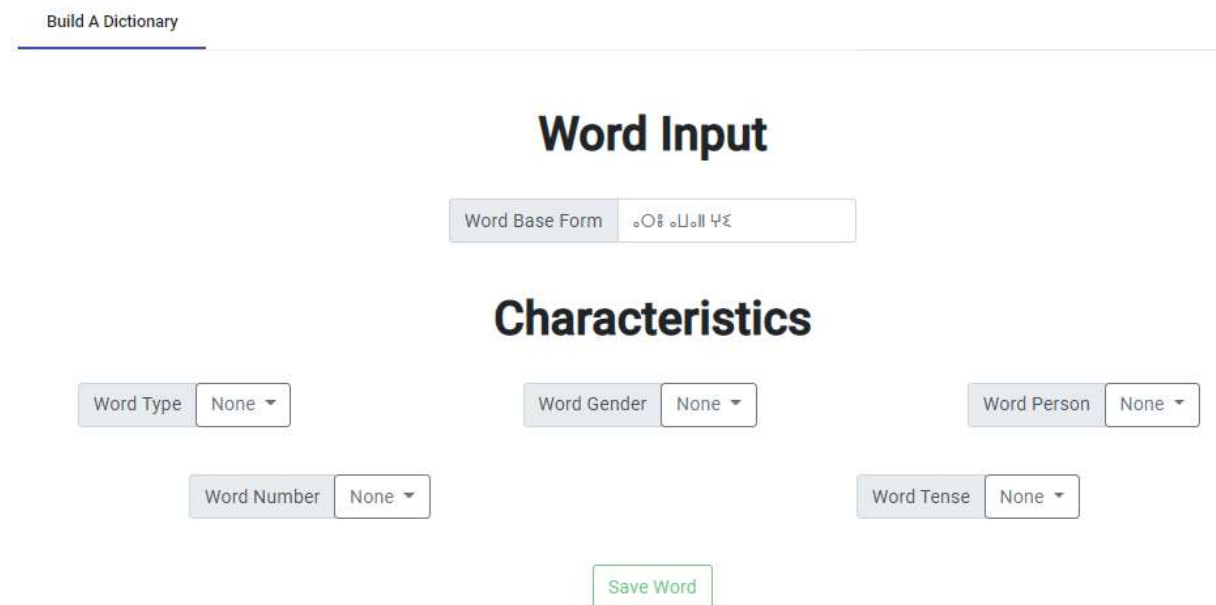
When the system returns a set of generated examples equal to the number they requested, each generated sentence is followed by two icons, a green check, and a red cross. These icons serve as feedback to the system on its generated sentences.

If a sentence generated by the system is grammatically correct, the user is going to click on the green check, sending the sentence back to the system, along with a label that marks the sentence as a correct one. The system then receives the sentence along with its label, updates its hypothesis by feeding it this new example, then pushes the example in the training data set. The same process applies to the incorrect examples, every system-user interaction is meant to further train the model to better understand the patterns that make up the structure of Tamazight.

The screenshots above make up the ensemble of possible interactions that the user can have with the system with regards to sentence generation and supervised learning. However, what happens if the user desires to enter a word that is not part of the dictionary of words we have? The answer is to just add it, by using the component explained below.

### 5.1.5. Updating the Dictionary

The user is given the possibility of typing in a sentence of their choice, following the corresponding pattern of every input field. However, this feature does not really allow the user to enter any word of their choice. The system has to know the characteristics of every word (Its gender, number, person...), and for that, the word entered by the user, as well as its part of speech, have to be present as part of the dictionary of words. However, the possibility of the user knowing more words than the system already does is not off the table, and is a very possible event. That's why, I have decided to give the opportunity for the user to populate the system's dictionary with new words from their library through a dictionary building page. This page includes an input field where the user can type their word in Tifinagh characters, then use the following fields to determine the characteristics of the word they are adding. The dictionary building page initially looks like this:



The screenshot shows a web interface titled "Build A Dictionary". Below the title is a horizontal line. The main content area is divided into two sections: "Word Input" and "Characteristics".

**Word Input**

There is a label "Word Base Form" followed by a text input field containing the Tifinagh word "ⵓⵔⵉ ⵎⵓⵎⵉ ⵙⵉⵔⵉ".

**Characteristics**

Below the "Characteristics" heading, there are five dropdown menus arranged in two rows:

- Row 1: "Word Type" (None ▼), "Word Gender" (None ▼), "Word Person" (None ▼)
- Row 2: "Word Number" (None ▼), "Word Tense" (None ▼)

At the bottom center, there is a green button labeled "Save Word".

Figure 16: Layout of the Word Addition Form

The word addition form requires the user to fill in all the required fields, before the Save Word button is enabled and can be pressed.

Below is a scenario of what happens when we fill in all the fields, and add a word to our dictionary:

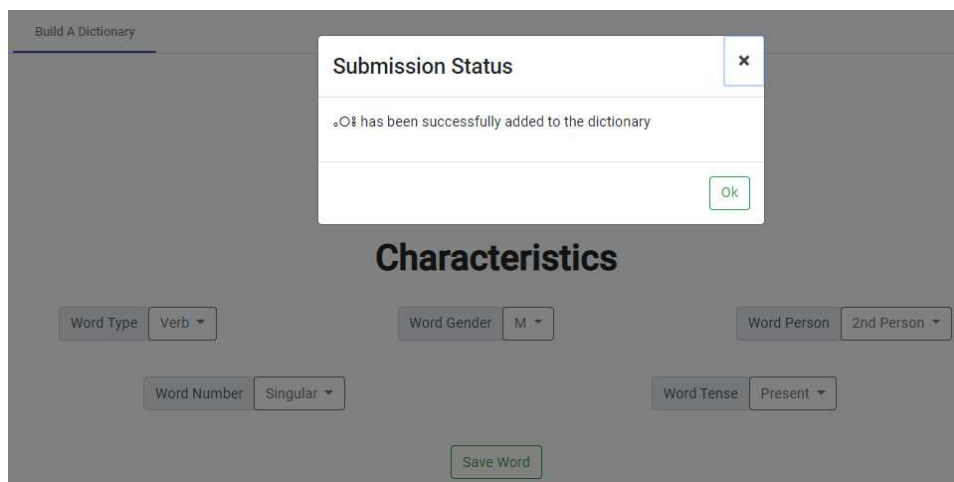


Figure 17: Word Submission Modal

The word we added is the verb “Write”, which is pronounced “Arou” in Tamazight, and written as “ⵝⵓⵔ” in Tifinagh letters.

After completing the fields, and clicking on the Save Word button, a modal appears, which confirms the addition of the word to the dictionary.

Before adding the word to the dictionary, we see in the screenshot below that the word never existed in the autocomplete:



Figure 18: Autocomplete Returning No Results on Search



After successfully creating the word, it is time to go back to the initial Example Generation page, and using the newly added word in a sentence to send to the system. Our example above can be found in the autocomplete list immediately after the addition as follows:

---

## Pattern

+ - 

Verb ▾

 +

## User Example

◦◦|

◦◦  
~~~~

Submit Correct Example

Submit Wrong Example

Generate Example

Figure 19: Word Successfully Added to the Autocomplete List

When the word is added to the dictionary, the user can expect good chances of the new word being used by the system as well, when it is working on generating new sentences after its training process.

## 5.2. Backend Design

### 5.2.1. Training Set Structure

Data, as they say, is the new oil, it is a very important component of every project, research, or analysis that we want to conduct. That is why, every piece of data that we can possibly gather from users is to be collected, analyzed, and stored for future use, without losing any piece of data provided.

For my project, data comes in the form of patterns, sentences, and words. This data is later used to train my model on hypotheses that structure the Tamazight language. However, the way my model is trained, which can be

found in later paragraphs, every single pattern has its own training process. Patterns are independent from each other, and every pattern requires a unique combination of agreements between its words, which make up the structure of a correct sentence. For example, the following pattern: **Common Noun, Verb, Common Noun**, if trained correctly, would require full agreements between all words in terms of gender, number, and most of the other characteristics, when it wants to form a correct sentence. However, sentences that follow the pattern: **Pronoun, Verb, Common Noun**, only require the first and second word to agree in terms of gender, number, person, and other characteristics. However, the third word of the sentence is mostly independent of the other two, and can take any form it wants, as long as it is of type **Common Noun**.

Now that we know that all patterns are independent, it is only logical to think of separating every pattern from the other, and having each pattern train on its own hypotheses, independently of all the others. In order to do that, I have created a system that, given a new pattern that it has never seen before, would create a separate training set for it, and save it as an independent file. Training sets are saved as CSV (Comma-Separated Value) files, and that's because Python has a great ability to deal with such files and store their values in appropriate data tables for their training processes.

In order to facilitate the access to my files, I have created a special naming system, which would allow the fastest access to the pattern I need through its corresponding data set. When provided with a pattern, I map every element of the pattern with two unique letters that make it different from all the other pattern elements. Then, I take the components of my pattern, then I concatenate all the mappings of its components to form a unique string, which serves as a unique name to the training set file. For example, given the following pattern: **Common Noun, Verb, Common Noun**, the name of the training data set would be: **CNVCN.csv**. That way, this file is guaranteed to only ever refer to that corresponding pattern. A very important advantage of this strategy is that, given a pattern, if I want to check whether it is part of an existing file I have, I do not have to loop

through all of the files I have till I find the pattern. I can instead simply pick up the name string, and see if the file exists or not.

When my data set is created, and after populating it with different examples given by the user, the whole thing is stored in a CSV file, which looks as follows (Pattern used: **Common Noun, Verb, Common Noun**):

|   | A         | B      | C         | D     |
|---|-----------|--------|-----------|-------|
| 1 | CN        | V      | CN        | Label |
| 2 | ⵍⵉⵎⵎⵉⵏ    | ⵜⵓⵔ    | ⵜⵓⵎⵎⵉⵏⵜⵓⵔ | Yes   |
| 3 | ⵜⵓⵎⵎⵉⵏ    | ⵜⵓⵔ    | ⵍⵉⵎⵎⵉⵏⵜⵓⵔ | Yes   |
| 4 | ⵜⵓⵔ       | ⵜⵓⵔ    | ⵍⵉⵎⵎⵉⵏⵜⵓⵔ | No    |
| 5 | ⵜⵓⵔ       | ⵜⵓⵔ    | ⵍⵉⵎⵎⵉⵏⵜⵓⵔ | Yes   |
| 6 | ⵍⵉⵎⵎⵉⵏ    | ⵜⵓⵔ    | ⵜⵓⵔⵍⵉⵎⵎⵉⵏ | Yes   |
| 7 | ⵜⵓⵔⵍⵉⵎⵎⵉⵏ | ⵍⵉⵎⵎⵉⵏ | ⵜⵓⵎⵎⵉⵏⵜⵓⵔ | No    |
| 8 |           |        |           |       |

Figure 20: Training Data Set Structure

As shown in the screenshot above, every sentence is written in Tifinagh alphabet, with every row representing an example, and every column representing a part of speech of the full pattern. The last column is the label of every sentence, which takes the form of a Boolean, this refers to the grammatical correctness of a sentence, where **Yes** refers to a correct sentence, while **No** refers to an incorrect one, it is then the system's job to figure out reasons for these labels.

### 5.2.2. Dictionary Structure

As explained in the User Interface section of this report, the system does not only receive input from the user as patterns or sentences to train on. It is also required by the user, as part of its training process, to generate sentences at random, following a given pattern by the user himself. The system may have ready-to-use hypotheses that form the agreement system with which it can generate sentences. However, it still needs words to

generate these sentences. For that purpose, we have created a dictionary of words, along with their characteristics, for the system to choose from.

The system builds its hypotheses based on word agreements in terms of gender, person, number, aspect, tense, and other characteristics, which are then applied to a pattern in order to know which words to choose for its sentence building. For that reason, the dictionary that it is provided with does not only contain words, but also multiple characteristics of every one of those words, precisely, all the characteristics that the system builds its hypotheses on.

Below is a screenshot which shows the dictionary structure of an ensemble of common nouns, along with their characteristics:

|    | A        | B         | C     | D   | E      | F    | G     | H   | I      | J   | K     |
|----|----------|-----------|-------|-----|--------|------|-------|-----|--------|-----|-------|
| 1  | BASE     | FULL      | MNG   | POS | TYPE   | PERS | TENSE | GEN | ASPECT | NUM | STATE |
| 2  | oOXoX    | oOXoX     | man   | N   | common | 3    |       | m   |        | s   | free  |
| 3  | oOXoX    | %OXoX     | man   | N   | common | 3    |       | m   |        | s   | annex |
| 4  | oOXoX    | %OXoXl    | men   | N   | common | 3    |       | m   |        | p   | free  |
| 5  | oOXoX    | %OXoXl    | men   | N   | common | 3    |       | m   |        | p   | annex |
| 6  | oOXoX    | %OXoXl    | men   | N   | common | 3    |       | m   |        | p   | free  |
| 7  | oOXoX    | %OXoXl    | men   | N   | common | 3    |       | m   |        | p   | annex |
| 8  | +oCEE%E+ | +oCEE%E+  | woman | N   | common | 3    |       | f   |        | s   | free  |
| 9  | +oCEE%E+ | +oCEE%E+  | woman | N   | common | 3    |       | f   |        | s   | annex |
| 10 | +oCEE%E+ | +oCEE%E+l | women | N   | common | 3    |       | f   |        | p   | free  |
| 11 | +oCEE%E+ | +oCEE%E+l | women | N   | common | 3    |       | f   |        | p   | annex |
| 12 | %G%OO%   | %G%OO%    | boy   | N   | common | 3    |       | m   |        | s   | free  |
| 13 | %G%OO%   | %G%OO%    | boy   | N   | common | 3    |       | m   |        | s   | annex |
| 14 | %G%OO%   | %G%OO%l   | boys  | N   | common | 3    |       | m   |        | p   | free  |
| 15 | %G%OO%   | %G%OO%l   | boys  | N   | common | 3    |       | m   |        | p   | annex |
| 16 | +oOo+    | +oOo+     | girl  | N   | common | 3    |       | f   |        | s   | free  |
| 17 | +oOo+    | +oOo+     | girl  | N   | common | 3    |       | f   |        | s   | annex |

Figure 21: Dictionary Structure of Common Nouns

As shown in the screenshot below, every row represents a word along with the characteristics that define it, and every column represents a characteristic of this word.

Characteristics of a word include: base form, form after applying all the characteristics, meaning in English, Part of Speech, Type, Tense (For Verbs), Gender, Aspect (For Verbs), Number, and State.

We notice that the characteristics are not all specific to common nouns, and that some of them are not even supposed to be common noun

characteristics (Tense and Aspect), but that is intentional. The training process of the model requires all words to have a uniform combination of characteristics, either filled or left empty, for the purpose of having a uniform hypothesis format for all the characteristics and patterns that I have, for the sake of simplicity, clarity, and ease of use and implementation.

### **5.2.3. Training Process**

The goal of my project is to build a system that performs supervised learning on user-generated patterns and sentences, and follow the training process with an automated generation of sentences based on what it learns. The sentences that the system generates have to be grammatically correct, i.e. ones that follow the grammar rules that structure a syntactically correct sentence in Tamazight.

Following the English language, “I want to eat popcorn” is a grammatically correct sentence, whereas “I wants to eat popcorn” is a flawed sentence because of the word “wants”. What makes this sentence incorrect is the fact that, in English, when a determinant is followed by a verb, they have to agree in terms of person and number. “I” is a pronoun of characteristics singular and 1<sup>st</sup> person, however, “wants” is a singular verb of the 3<sup>rd</sup> person, and that’s what makes this sentence incorrect.

Based on this logic, the goal of my training process is to create a hypothesis, or a version space, that would contain all of the agreements needed for a correct sentence, given a certain pattern for it.

A sentence is a group of words of a certain length, where each word can be assigned a unique index going from 1 to the length of the sentence. For the sake of memory optimization, I have decided that these indices would be the word indicator in my hypothesis space, where the first word of a sentence is given index 1, the second is given index 2, and so on.

Each word has several characteristics which we are to extract our hypotheses from, such as gender, person, number... However, not all characteristics have the same agreement policy. For gender, the first word

may only need to agree with the second and fifth word for a sentence to be correct, whereas the first word must agree with the second, third, and fourth for the person and number characteristics. This tells us that, just like all patterns are independent from each other, all characteristics of each pattern are, to some extent, independent from each other as well.

Using this information, I have decided to develop a separate hypothesis space for every characteristic alone, where I would have a gender hypothesis, a person one, a number one, and so on.

The examples I will use will solely follow the Gender hypothesis in Tamazight sentences. Given our usual example so far of the following pattern: **Common Noun, Verb, Common Noun**, and our first positive example being: **ⵎⵉⵎⵓⵏ ⵜⵉⵔⵉⵙⵜ ⵉⵎⵓⵏ**, which translates to “My sister is a doctor”, my initial specific and general hypotheses will look like this:

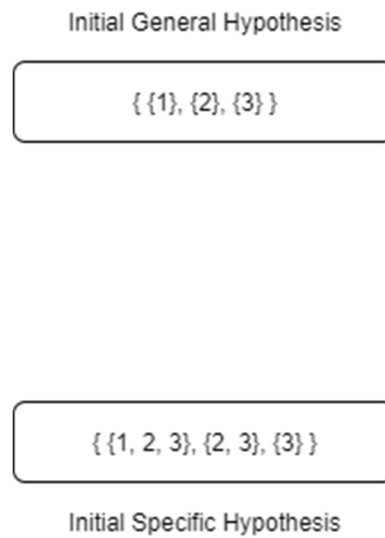


Figure 22: Initial Structure of Training Hypotheses

As defined in the Version Space part of this report, the general hypothesis has to be initialized to the null description, which is a description of the sentence structure that matches everything. The specific hypothesis, on the other hand, has to be maximally specific, but still correct, so we initialize it to include the first positive example in the training set.

I have written the specific hypothesis in a way that avoids redundancy, where every cell of the hypothesis only includes indices of the words that follow the current one, you will never find a second cell containing the number one, because the first cell will already have included the number two if the agreement between the two words were mandatory.

The meaning of the numbers inside the hypothesis are the indices of the words. For example, in the initial general hypothesis, the null description is written as:  $\{\{1\}, \{2\}, \{3\}\}$ , which indicates that the first word should only ever agree with itself in terms of gender, and similarly for the following two words. This does not mean that the first word must not agree with the other two, but it only means that the first word must have the same gender as itself, whereas it may agree or disagree with the other two, whichever it is would not affect the correctness of the sentence in any way, this hypothesis format guarantees that all possible description of the language are included, since all words agree with themselves in all characteristics. For the specific hypothesis, the gender happens to be mutual between all words, where the first word must have the same gender as the second and third words, hence the reason the first cell has all three numbers, indicating that the first word must agree with itself, the second, and the third word for the sentence to be correct.

For the same pattern mentioned before: **Common Noun, Verb, Common Noun**, if we were given a negative example, say: **UM+C. X.I .EΘξΘ**, which translates to “My sister are a doctor (male)”, we notice the differences between this example and the following one. Following the definition of the Version Space algorithm, this example should be used to adjust our general hypothesis, and minimally generalize it to exclude this possibility of a gender disagreement.

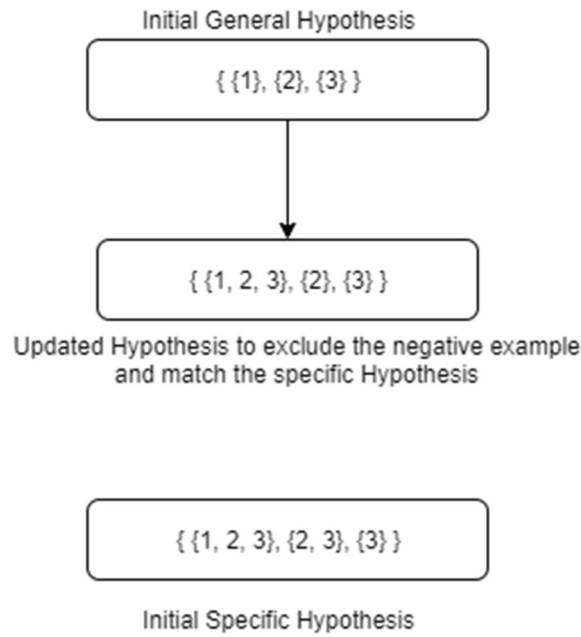


Figure 23: Hypothesis Update on Negative Example

The second and third word in the negative example agree in terms of gender, which is something that is also part of the specific hypothesis, so we do not update that part of the hypothesis until we confirm how correct that is through future examples. However, the first word disagrees with both of the following words in the negative example, but the positive one shows that the first word must agree with both words that follow, which is why we update the general hypothesis to exclude the possibility of disagreement, and match the specific hypothesis.

For this specific pattern, we must expect to keep going through examples, generalizing specific hypotheses on positive examples, and specializing general hypotheses on negative ones, until we reach one final gender agreement hypothesis between words, in the following form:



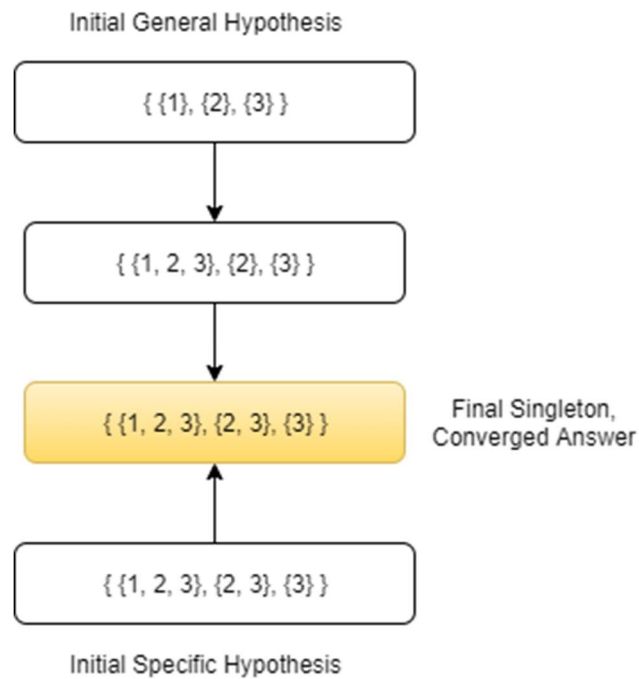


Figure 24: Converged Hypothesis to Final Answer

When the examples in the training set are all exhausted and used for the training process, it is expected to find that the general and specific gender hypotheses have converged to one unified hypothesis, which defines our final answer. However, sometimes it's not expected to have a converged answer, and that's because the model still needs further training. This is highly expected because my model trains on the go, and as it receives further examples from the user.

If the system has not yet fully trained, but is still requested by the user to generate sample sentences, the system must not refuse, and must find a way to still generate examples for the user. The way this is implemented is, given a set of incomplete hypotheses, we are to generate one temporary hypothesis which encapsulates what we have learned so far, which may or may not be fully correct. The way we do this is by assumption. We know that our specific hypotheses are the ones that are mostly based on correct examples, and are therefore our most likely way of generating a correct sentence. So what I do is, going through all of my specific hypotheses, I take the hypothesis for a word that has the most number of agreements within, because that's my most correct example in the training set so far:

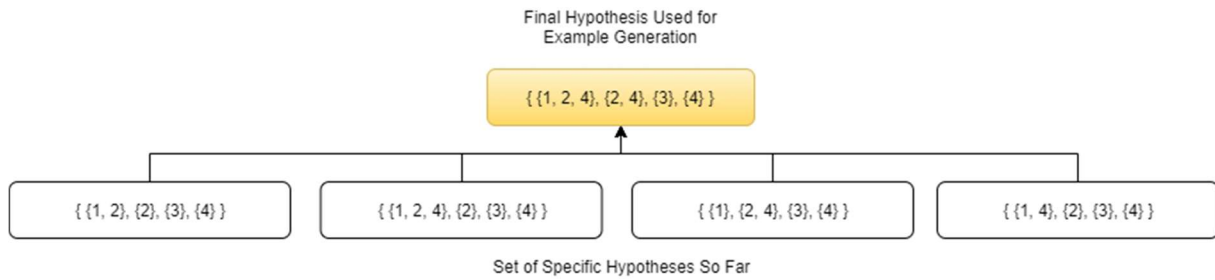


Figure 25: Combining Specific Hypotheses for Sentence Generation

This unified hypothesis may or may not be a way to generate correct examples for the given pattern of size 4 as shown on the figure. However, it is simply our best chance to do so for now. It is even better if it generates wrong examples from this hypothesis, because that only shows that there is still room for further training, which will be provided by the user as feedback on the sentences that the system sends based on that unified hypothesis.

#### 5.2.4. Sentence Generation Process

Every sentence has a set of predefined characteristics: They follow a specific pattern to their structure (E.g. **Common Name, Verb, Common Name**), They are of a specific length, which is equal to the number of elements in the specified pattern, and, from a grammatical perspective, they follow a certain set of grammatical rules which makes of them correct sentences.

All of the conditions above have to be presented to my system in order for my generation algorithm to take place. Given a pattern of a certain length, my system looks up the data sets that it has stored in order to find one that is trained based on the pattern given by the user. Then, the system looks up the set of hypotheses that it has generated from the training process of that specific pattern. If the hypotheses generated have already reached their singleton stage (converged to an answer), then it just uses that hypothesis to generate the sentences. If the hypotheses found did not yet converge, then we use the method specified in the last figure above, which combines the specific hypotheses into one unified temporary one for the purpose of the sentence generation.

If the system ends up not finding any data set for the specified pattern, it warns the user about it, and asks for examples of sentences that follow that pattern which it could train on. But if the system succeeds in finding a data set, and in the extraction of the hypotheses it generated from the training process, then the sentence generation process begins.

### DFS Pseudocode:

For my generation algorithm, I have used a **Depth-First Search**<sup>1</sup> traversal algorithm. Below is the pseudocode for my traversal, and for the sake of simplicity, I have only worked with the Gender feature in demonstrating both the pseudocode. I will also only use the Gender example for the following demonstrations of this algorithm.

---

**Algorithm 1** Depth First Search for Automated Sentence Generation

---

```

procedure DFS(indexSoFar, SentenceSoFar)
  Initialize all features (gender, person, number...) to an empty list
  Perform a random shuffle to the data set to avoid multiple use of words

  if indexSoFar is equal to PatternSize then
    Append SentenceSoFar to Sentences to Return
    return

  GenderFeature  $\leftarrow$  "" ▷ We will only use the Gender example

  for each indexofWord in SentenceSoFar do
    if indexSoFar must agree with indexOfWord then
      ▷ GenderOfWord is Gender of word reached by loop
      GenderFeature  $\leftarrow$  GenderOfWord
      Break Loop

  for every WordReached of DictionaryOfWords do
    TemporarySentence  $\leftarrow$  SentenceSoFar
    ▷ GenderOfWord is Gender of word reached by loop
    if GenderFeature is same as GenderOfWord then
      Append WordReached to TemporarySentence
      DFS(indexSoFar + 1, TemporarySentence)

```

---

Figure 26: DFS Traversal for Sentence Generation

<sup>1</sup> DFS is an algorithm that is used to traverse tree structures by following each node, child by child, until a leaf is found, then recursively going back parent by parent and exploring their children, until every node in the tree is traversed at least once.

This DFS function is a recursive one, meaning that the process of its implementation can be represented as a tree, but before we draw that hierarchical representation, let us go through what each line of code means: For the function parameters: `indexSoFar` is the index of the word we are currently trying to choose, and `SentenceSoFar` is the list of the chosen words, and that list is of size = `indexSoFar`. The base case of the DFS is when we have entirely built our sentence, and that is achieved by building a number of words equal to the size of the pattern sent to the system by the user, when that condition is satisfied, our `SentenceSoFar` is appended to our list of sentences to return to the user.

### Recursive Tree Representation:

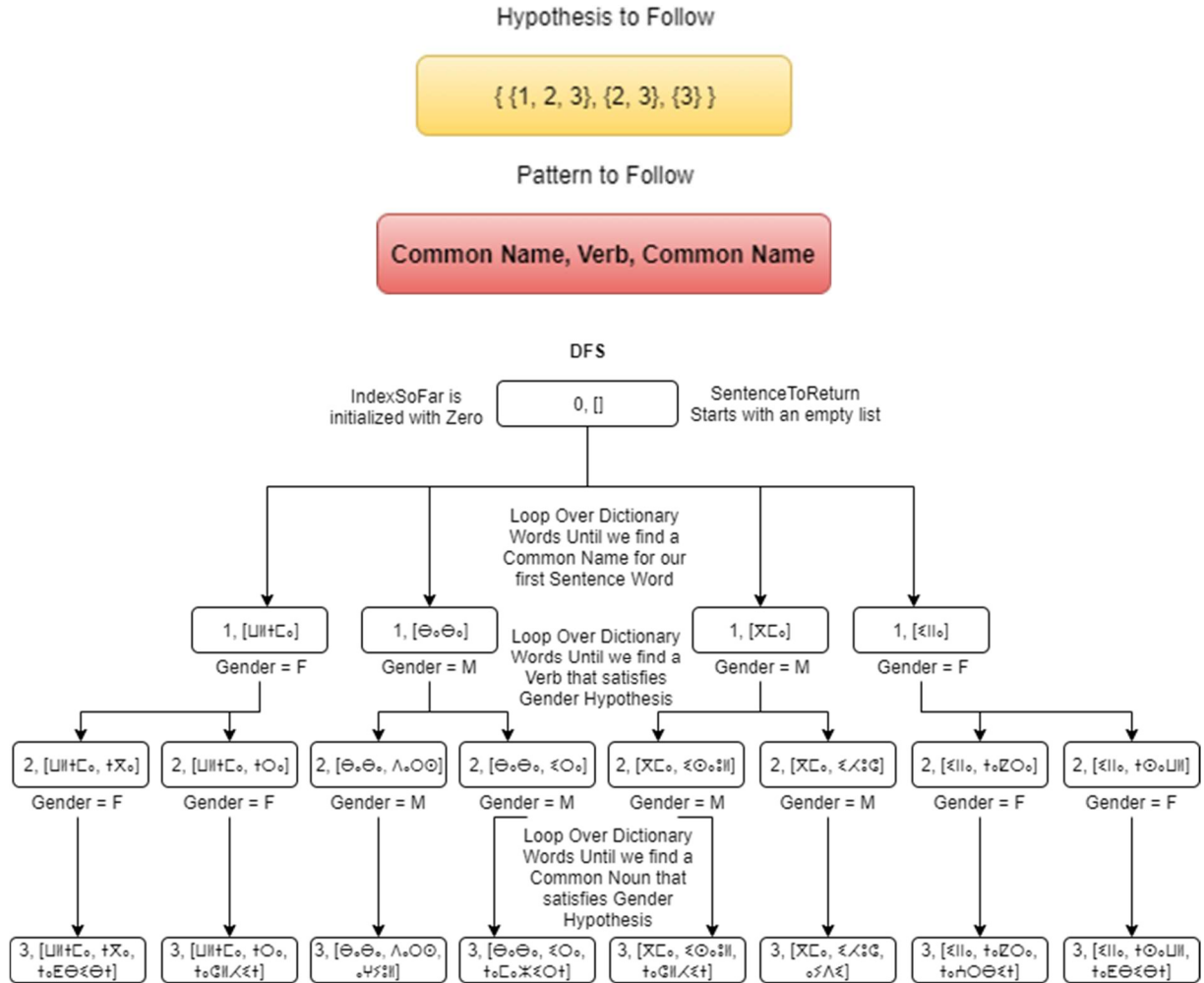


Figure 27: Recursive Tree of the Sentence-Generation Algorithm

The figure above is a representation of the Depth-First Search traversal algorithm, which loops through the dictionary of words until it finds a word which satisfies the conditions of the word we want to build in the current iteration. Initially, we are looking for a Common Noun, so the DFS calls itself recursively 4 times, because it found 4 Common Nouns (ⓂⓂⓈⓈ, ⓈⓈⓈⓈ, ⓈⓈⓈⓈ, ⓈⓈⓈⓈ) in the dictionary. Then, for each of those words, we look for a verb which satisfies the hypothesis above. Our hypothesis states that the gender of the second word must agree with that of the first word, so, for each of those words, the system has found 2 Verbs, which agree in gender with the first word of the respective lists. There is a very low chance of the words being repeated in two sentences of the same DFS, because at the beginning of every function call, we perform a random shuffle to the dictionary, as shown in the pseudocode in Figure 26. The verbs that the system has found are:

- For ⓂⓂⓈⓈ (Ultma), it returned ⓈⓈⓈⓈ (Tga) and ⓈⓈⓈⓈ (Tra)
- For ⓈⓈⓈⓈ (Baba), it found ⓈⓈⓈⓈ (Dars) and ⓈⓈⓈⓈ (Ira)
- For ⓈⓈⓈⓈ (Gma), it found ⓈⓈⓈⓈ (Isawl) and ⓈⓈⓈⓈ (Ihush)
- For ⓈⓈⓈⓈ (Inna), it found ⓈⓈⓈⓈ (Taqla) and ⓈⓈⓈⓈ (Tsawl)

All of the words returned agree with the common name in terms of gender, which makes the sentence grammatically correct according to the used hypothesis. Similarly, for the third word to be built, the system only returns Common Nouns that it finds in the dictionary, which agree with both the first and second words of the sentence built so far.

### **How to Avoid Repeatedly Returning Same Sentence to the User:**

When the Sentence-generation algorithm is done running on all possible combinations of the given pattern, it is time to return the sentences found to the user. Given the number of sentences that the user desires as N, the obvious solution is to just loop over the sentences generated by the DFS, and just take the first N sentences and return them to the user. However, as the user requests more sentences multiple times from the system, there are high chances that the sentences generated will be the same at some point, and that the same sentence would be returned multiple times to the user.

One way to resolve this issue is to take advantage of the fact that every generated sentence sent to the user requires their feedback. Keeping that in mind, all we need to do is to keep track of all the sentences returned by the user, both the ones manually typed by them, and the ones returned with a verdict, after the system automatically generated them. Once we have that stored in our training data set, we can take the ensemble of sentences generated by the system, loop through them, and check for each sentence, its existence in the training set. If the sentence already exists, then it must be skipped, otherwise, we append it to the list of chosen sentences. We repeat that until we have a new list of  $N$  chosen sentences, then return those to the user for a final verdict.

## **6. Final Remarks**

### **6.1. Limitations and Challenges**

While developing this project, I was faced with several challenges. This project was my very first experience with Natural Language Processing and its applications, I therefore had to read about a lot of tools and algorithms, before knowing what to choose from and what to apply to my project. The Version Space algorithm may be the best algorithm for the purpose of my project; however, I could not find any previous implementations of this algorithm, especially for the purpose of sentence generation of my project. For that reason, the best solution I could find is to read about the algorithm, understand how it works, see how I can adapt its use to serve the purpose of my project, and implement the algorithm from scratch using my own logic. The second challenge we faced was that of the language to get the system to learn, we had discussed the possibility of French, because we would have more resources for the dictionary, and could use more programmed tools that help with the processing of natural languages in Python. However, we have decided to go for Tamazight, because the need for such a system in Tamazight is much wider than the need of it in any other language, especially in Morocco. This choice was made based on the increasing interest that people have for Tamazight, versus the decreasing number of people that actually use it. This choice came with its challenges as well, Tamazight is not

only a different spoken language, it is also one that has a completely different alphabet to it. The positive side to this is that the users meant to use this system are proficient people in Tamazight, who are familiar enough with the Tifinagh alphabet. However, today's computers do not have the Tifinagh alphabet embedded in them, so the challenge was to find a way to give the users the ability to write full sentences in Tamazight, without having to download any external keyboards ("Keyman Desktop 12.0", 2019).

## 6.2. Future Work

This project aims to teach a language to its users through user-system interaction. For that, the system will have to be able to understand the input of the people using it, be able to make sense of it, and learn from it. For the scope of this project, all of these features were achieved, but only for the syntactic and grammatical rules of the sentence. When it comes to the meaning of the sentences generated by the system, there are very high chances that the examples are semantically meaningless (Possible generated sentence to be labeled as **correct: My dog is a cat**). For this imperfection to be avoided, I will have to read more about Semantic Labeling, and go deeper into the analysis of semantics in Natural Language Processing. This is beyond the scope of this capstone project, but with the agreement of my supervisor, this development process of this project will continue, and I will make sure to include this feature, along with other features to ensure a faster and more accurate learning process for the system, and a better experience for the user. Once that is achieved, my system will be able to use the learnt hypotheses to generate examples that it can turn into exercises, which will make the system reach its full potential and initial educational purpose of teaching interested people how to formulate syntactically, grammatically, and semantically correct sentences in Tamazight.

## References

- Angular 8. (2019). Retrieved 25 November 2019, from <https://angular.io/>
- Asegzawal Tamaziyt-Tanglizt | Tamazight-English Dictionary. (2019). Retrieved 25 November 2019, from <http://asegzawal.com/english/#>
- Flowchart Maker & Online Diagram Software. (2019). Retrieved 25 November 2019, from <https://www.draw.io/>
- Foundation, N. (2019). Node.js. Retrieved 25 November 2019, from <https://nodejs.org/en/>
- Jurafsky, D., & Martin, J. (2014). *Speech and language processing*. [India]: Dorling Kindersley Pvt, Ltd.
- Keyman Desktop 12.0. (2019). Retrieved 25 November 2019, from <https://keyman.com/desktop/>
- Mitchell, T. (1997). *Machine learning*. New York: McGraw Hill.
- Otto, M. (2019). Bootstrap. Retrieved 25 November 2019, from <https://getbootstrap.com/>
- PESTEL Analysis (PEST Analysis) EXPLAINED with EXAMPLES | B2U. (2019). Retrieved 25 October 2019, from <https://www.business-to-you.com/scanning-the-environment-pestel-analysis/>
- Welcome to Flask — Flask Documentation (1.1.x). (2019). Retrieved 25 November 2019, from <http://flask.palletsprojects.com/en/1.1.x/>
- Welcome to Python.org. (2019). Retrieved 25 November 2019, from <https://www.python.org/>