qT3: An App for the Algorithmic Deduction of Arabic Verbs

CS 701 Final Project, Middlebury College, Fall 2021

Leah Smith
Middlebury College
lesmith@middlebury.edu

Sabrina Templeton
Middlebury College
shtempleton@middlebury.edu

ABSTRACT

Arabic as a language has predictable patterns that lend itself to algorithmic deduction. As Computer Science students and Arabic learners ourselves, we recognized the algorithmic grammar of the Arabic language and wanted to explore ways of combining the fields. In this project, we set out to create an app that would assist Arabic students in their understanding of Arabic verbs, without taking away their opportunity to learn by providing a direct translation.

The result of our project is a functional iOS app which takes an Arabic verb as input and presents the user with information about the form, root, prefixes, and suffixes. It also provides information on the verb's features: tense, person, mood, number, and gender. Our algorithm is able to deal with irregular verb cases as well. We also implemented a test suite to test various parts of our algorithm pipeline and ensure functionality for special cases. Additionally, the app deals with and displays various cases of ambiguity, which arise because the input is meant to be a verb in any form, conjugation, with any number of affixes, and with no short vowels written. In such cases of ambiguity, the app returns all potential forms or features, allowing the user to decide which result is most relevant to their case. The app encourages users to validate the results and discover word meaning by automatically providing links to Wiktionary entries for each result. Tags are displayed to provide the user with information about prefixes or suffixes that the algorithm removed, as well as flagging a word if it is potentially irregular. Our app fills a gap in current technologies by catering to beginner students and providing them the most helpful information for their learning. It also exemplifies a new method of deducing this information in a purely algorithmic way.

1 INTRODUCTION

We were inspired in particular by the algorithmic nature of Arabic verb forms and conjugations, which follow strict and predictable patterns. Additionally, we recognize the challenge posed by current Arabic resources for beginning Arabic students. The use of an Arabic or Arabic-English dictionary first requires knowledge of the root of the Arabic word. For beginning Arabic students, this can be a huge challenge to look up verbs, since one must be able to identify the three-letter root of the verb by omitting prefixes and suffixes, reverse engineering the conjugation pattern of the verb, and recognizing the verb form in order to parse it correctly. This is not to mention the numerous exceptions for irregular verbs that exist. Our app is able to perform these tasks algorithmically, and returns all the possible information to the user, something which has never been done. Our project is novel both in method and in the accessibility of the final application.

In this paper, we will briefly cover the background of Arabic that is necessary to understand our work. We will talk about the problem statement that our work addresses, and review some related technologies and research. This paper then covers our methods, going through the pipeline of how our algorithm works as well as a brief explanation of the technology pipeline which results in our mobile application. We will then go over the results, which include screenshots from our current mobile app prototype as well as an overview of our testing suite. Finally, this paper ends with a discussion of current limitations as well as ideas and direction for future work.

2 BACKGROUND

Arabic verbs have a number of characteristics that overlap with, and set them apart from, other languages. In this section, we will go over what is necessary to know about Arabic verbs in order to understand the challenges our project presented and how our app works, assuming no prior knowledge of Arabic.

2.1 Verb conjugations

Arabic verbs have conjugations just like verbs in many other languages do. Arabic conjugations reflect the tense and mood, and they must also match the subject in person, number, and gender.

Arabic has two tenses: imperfect and perfect. For the purposes of this paper, we refer to these as "present" and "past" respectively, as this is how they are often construed into English. There is also a prefix in Arabic, the letter "s", which shows that a present-tense verb is actually in the future tense. The key thing to know here is that a verb can *only* have the future prefix if it has been conjugated in the present tense. Finally, it is important to note that Arabic conjugation patterns add letters to both the beginnings and ends of verbs. This is different from many languages that only conjugate verbs by adding letters to the end of the word; English is a good example of this: I walk, he walk*s*.

Arabic has three moods: indicative, subjunctive, and jussive. The context of a sentence determines the mood of the verb, which in turn affects the conjugation pattern of the verb.

Arabic, like English, uses the first person, second person, and third person. First person refers to "I", "me", "we", or "us". Second person refers to "you" or "you all". Third person refers to "he", "she", "it", or "they."

Arabic has singular and plural, but it also has a dual form. This means that the verb is conjugated differently depending on whether it refers to one person doing the action, two people doing the action, or three or more people doing the action.

Finally, Arabic has two genders: masculine and feminine. Not all conjugations are affected by gender. For example, verbs conjugated in the first person (e.g., *I* do something) do not reflect the gender

of the speaker, but verbs conjugated in the second person and third person do.

2.2 Verb forms

The biggest thing that sets apart Arabic verbs from verbs in many other languages is the structured verb forms, as well as the triliteral root system. All Semitic languages use the triliteral root system, meaning there are three letters in every word that reflect the core meaning of that word. In Arabic, these three root letters have templates (comprised of other letters) applied to them to create new meanings that are still connected to the root's meaning.

Typically, the triliteral root is made up of only consonants. The verb forms take these consonants and place specific vowels and consonants before, after, and between the root letters to create a new meaning. For example, the name of this app is qT3. This is the Form I verb of the three letter root q-T-3, meaning "to cut". The Form II pattern simply doubles the middle root letter of the base verb, so the Form II verb of q-T-3 is qTT3, meaning both "to slice" and "to cut someone". The Form VI pattern adds a "t" to the beginning of the root and an "a" between the first and second root letters, so the Form VI verb of q-T-3 is tqaT3, meaning "to separate". As a final example, the Form X pattern adds an "ist" before the root letters, turning q-T-3 into "istqT3". This word means "to deduct," which is ultimately the goal of our project.

The verb's form affects its meaning. One can think of it like this: as the number of the verb form increases, the abstraction of the meaning also increases. In this sense, Form I is the "purest" or most basic meaning of any three-letter root, and Form X is the most abstract. Each form has a particular meaning associated with it that is applied to the Form I meaning. Figure 1 shows the verb forms and their associated meanings.

There are 15 verb forms in Arabic, marked with Roman numerals [8]. However, only 9 of these are commonly used, and only 10 are even taught to Arabic students. The 5 additional verb forms are not taught, and many native speakers do not know they exist, as they were already rare in Classical Arabic (used from the 7th to 9th centuries) and are even more rare in Modern Standard Arabic [7]. For this reason, our app only addresses the 9 most common verb forms: I, II, III, IV, V, VI, VII, VIII, and X.

Form IX is not in this list because although it is still used in Modern Standard Arabic, it appears very rarely and is only used in one particular case: color words. For example, Form IX would be used with the root meaning "red," to turn it into a verb meaning "to blush." Because color words are such an uncommon and particular case, we chose not to include Form IX in our algorithm.

2.3 Irregular Verbs

There are a number of exceptions to the "typical" verb in Arabic: weak verbs, geminated verbs, and phonological changes that affect the spelling of verbs. Each of these had to be accounted for in the algorithm, without the adjustments we made for them obstructing the app's ability to properly identify regular verbs.

Weak verbs. A weak verb is a verb which has a vowel as one of the root letters. There are three types of weak verbs, demarcated by the location of the vowel. If a verb has its vowel in the first position of the three-letter root, this is referred to as an *assimilated* verb. If the vowel is in the middle position, this is a *hollow* verb. If the vowel is in the final position, it is a *defective* verb.

The position of the vowel in the root is significant because weak verbs may drop or change the vowel depending on the form, conjugation, and position of the vowel. Therefore, assimilated, hollow, and defective verbs each must be handled in a different way by the algorithm.

Geminated verbs. A geminated verb is a verb which has the same letter twice in the root, in the second and third positions of the root. This is significant because in many instances, the doubled letters are combined into one *stronger* letter. In the case of our app, if we were to not account for geminated verbs with a special exception, then we would often have the shadda (a diacritic symbol which shows emphasis) marked as one of the root letters, which is incorrect.

Phonological changes in spelling. In some instances, the phonetic quality of a letter is impacted by the letter immediately preceding it.

As an example in English, the phrase "would you" is rarely pronounced with a separate "d" and "y" between the words. More often, native English speakers pronounce this phrase as though the "d" and "y" combine into the letter "j": "wuh joo". This is called phonetic assimilation [4].

In Arabic, a similar phenomenon happens when the first consonant is "stronger" than the consonant following it. This impacts the spelling of the word as well, unlike in the English example. Most significantly, this occurs in Form VIII because there is no short vowel separating the first root letter from the letter "t" following it. This means that Form VIII cannot be identified as easily, because the "t" which indicates that it could be Form VIII often presents as another letter. Figure 3 shows the list of spelling changes our algorithm needed to account for as a result of this.

3 PROBLEM STATEMENT

The objective of our project is to create an easy-to-use app that can parse Arabic verbs in any form, distilling the provided data into the fundamental information about the word.

The result of our project is an iOS app which takes input as any Arabic verb and presents the user with information about the form, root, prefixes, suffixes, and conjugation features of the verb. It also links to a Wiktionary page for each result.

4 RELATED WORK

Previous research in this field includes the work of Sembok et. al [6], which examines the previous stemming algorithm created by Al-Omari [1] and attempts to improve on it using a new approach. Sembok's approach essentially iterates through prefixes and suffixes to extract possible stems, using validation from some 800 rules about which prefixes and suffixes are viable. From there, the algorithm calculates the root by iterating through all possible stems produced by these iterations, and then validating the resulting root against a database of known Arabic roots. This approach is more brute-force as compared to our approach. Also, this approach does not retain any of the information that could be gained from the data that is thrown away, such as suffix meaning or verb form.

Form	Arabic	English	Meaning applied	Example: Form I vs. Given Form
Form I	فعل	f3l	Expresses the general verbal meaning of the root in question	Form I, clearest meaning of the root
Form II	فعّل	f33l	Often a *causative* version of Form I, or an *intensive* version of Form I	"to collect, gather" vs. intensive: "to amass, accumulate"
Form III	فاعل	fa3l	Usually gives an *associative* meaning to Form I; describes doing the action to or with someone else	"to work" vs. associative: "to deal with"
Form IV	أفعل	af3l	Usually a *causative* version of Form I	"to leave" vs. causative: "to expel, to evict"
Form V	تفعّل	tf33l	Often a *reflexive* version of Form II	"to gather" vs. intensive: "to congregate"
Form VI	تفاعل	tfa3l	Usually a *reflexive* version of Form III	Form III being "to deal with", vs. reflexive: "to deal with each other"
Form VII	انفعل	anf3l	Usually a *reflexive* or *passive* version of Form I	"to cut off" vs. passive: "to be cut off"
Form VIII	افتعل	aft3l	Often a *reflexive* version of Form I	"to gather" vs. reflexive: "to meet"
Form X	استفعل	astf3l	Often a *considerative* or *requestive* version of Form I	"to be far away" vs. considerative: "to consider something remote or unlikely"

Figure 1: Verb forms and their meanings. The right-most column shows possible meanings of Form I verbs as compared to the form given by the row; these are translations of real Arabic verbs in both forms, but the Arabic is not included in this diagram. Descriptions and examples taken from Arabic Desert Sky [2].

Figure 2: Example of a geminated word. Given the root letters "sh" + "d" + "d", the two "d" letters combine into one stronger "d" with a shadda symbol above it. This image actually depicts the word "shadda", a verb meaning "to make stronger."

Similarly, Buckwalter [3] implemented a morphology analysis algorithm which breaks out the potential prefixes, suffixes, and stem of a word. At each stage, Buckwalter validates the prefixes, suffixes, and stem combinations against a database of each and ensures that the combinations of prefixes and suffixes are valid. However, Buckwalter's algorithm only produces a conjugated stem of the word and does not attempt to identify the root or verb form from this stem.

Our approach is fundamentally different from these primarily research-based approaches in that ours is a user-centered app geared towards providing as much information as possible to a student. Other apps and technologies that exist to aid students fall into two main categories: dictionary apps and translation apps. Dictionaries and dictionary apps are effective for experienced Arabic students who have pre-existing knowledge of the language and are therefore able to correctly identify the root themselves in order to properly look it up in an Arabic-English dictionary. However, they are not very accessible to beginning students of the language who are not able to reliably get over this barrier to lookup. These students may then turn to the other currently available and popular option: translation apps such as Google Translate. However, these apps are often incorrect, which early students lack the knowledge to recognize. Furthermore, they do not provide enough information about the structure of the Arabic word to properly assist the student in their learning; translation apps merely produce an answer with no guidance on how that answer was determined. Our app fills this gap in existing technologies by taking a new approach which is more purely algorithmic and retains more information about the word than previous research has.

5 METHODS

In this section, we will go through step by step how our app and algorithm function. To see for yourself, the Github repository is linked here: https://github.com/leahets/qT3

5.1 Word class

To reflect the many attributes of Arabic verbs, our algorithm includes a Word class, in which each attribute gets filled with information as the object is passed through the algorithm pipeline.

The attributes include the following: *raw text, conjugated form, set of features, third past form, set of checked forms, root, form, prefix count, suffix count, list of possible prefixes, future boolean, weak boolean, invalid boolean, list of dropped prefixes, dropped suffix, hollow boolean, defective boolean, geminated boolean, assimilated boolean.*

First root letter	Pronunciation of letter	Changes ت (t) to....	Root of example word	Form VIII without spelling change	Actual Form VIII
ص	(ṣ)	ط	صدم	إصتدم	إصطدم
ض	(ḍ)	ط	ضرب	إضترب	إضطرب
ز	(z)	د	زرع	إزترع	إزدرع
د	(d)	دّ	دعى	إدتعى	إدّعى
ذ	(dh)	دّ	ذخر	إذتخر	إدّخر
ط	(ṭ)	طّ	طلع	إطتلع	إطّلع
ظ	(ẓ)	ظّ	ظلم	إظتلم	إظّلم
ت	(t)	تّ	تجر	إتتجر	إتّجر

Figure 3: Spelling changes as a result of Arabic phonological rules. Examples sourced from Elementary Arabic web page [5].

5.2 App interface

The application is accessed by the user by a iOS interface, developed in Swift using xCode. When the word is input by the user, it is run through the algorithm described below.

5.3 Pre-processing

After receiving the input, the algorithm first checks if the string contains any English letters. If so, it sets the form to English and returns an otherwise empty Word object with no other information.

From there, the algorithm creates all possible feature sets for any conjugation. In this paper, we use the term "feature set" to mean the group of attributes that contains tense, person, gender, number, and mood. A single feature set has each of these features accounted for exactly once. Overall, if we assume any combination of features is possible, then there would be 216 possible feature sets. However, not all of these feature sets are grammatically correct.

As stated before, first person conjugations do not reflect gender; therefore there is not a feature set for first person feminine or first person masculine. Rather, there is one feature set that is first person, non-gendered. As another example, past tense verbs do not reflect mood, so there are not 3 moods for each past tense feature set; there are only feature sets with past tense and no mood. These, among other exceptions, are why we chose to hard-code the feature sets rather than determine them algorithmically. This approach resulted in 54 feature sets in total and enabled us to confirm that our deconjugation algorithm was correct by allowing us to check that every feature set was accounted for.

To minimize the amount of hard-coding required, we created a `decode_features` function that takes a string and converts it into a Features object. The strings are 5 characters long, and each character symbolizes one of the features of a conjugation. For example, the string "r2m1s" would be converted into a Features object, where "r" signifies present tense, 2 signifies second person (you), "m" signifies masculine gender, 1 signifies singular, and "s" signifies that it is in the subjunctive mood.

5.4 Creating possible words

The first step in the algorithm is to take the given text and create every possible word that it could be by iteratively dropping prefixes and suffixes. There are a predetermined number of prefixes and suffixes that a verb can have attached to it, with the maximum being three prefixes and one suffix, although suffixes can be multiple letters in length. The `create_possible_words` function generates a list of strings that contain every combination between all affixes attached and no affixes attached. In this way, we are able to run different versions of the same input through the entire algorithm to see if they are valid. This is necessary because some words contain letters that match possible affixes, so the algorithm needs to test it with and without affixes to ensure that the correct combination is always tested.

For example, the word "safhm" would create four possible strings to be processed in the rest of the algorithm, since "s" is an appropriate prefix (meaning "will") and "hm" is an appropriate suffix (meaning "them"). To test all possible combinations of prefixes and suffixes, the `create_possible_words` function will produce the following list: safhm, saf, afhm, af. This can be seen in Figure 4.

After all possible words have been created, the algorithm then removes duplicates from the list while maintaining order. This is necessary because our `create_possible_words` function produces words in the order of likelihood, but can create duplicates if the word has no prefixes or no suffixes. Therefore, it is important to maintain the order of the list so that the app can properly display results that are most likely to be correct once the rest of the algorithm has run.

5.5 Pipeline

The next step is to run each word generated by the `create_possible_words` function through the "main" pipeline. The `pipeline` function is structured as follows:

- `check_invalid_preconjugate`
- `check_double_hamza`
- `deconjugate`
- `strip_fixes`

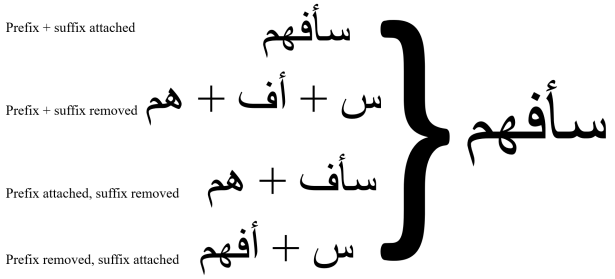

Figure 4: How the word "safhm" would be split up by the algorithm. In this example, the correct combination would be the last one, "s" + "afhm", because the correct root of the word is "fhm". However, it is impossible to algorithmically determine that this is more correct than the third option, "saf" + "hm". The qT3 app eliminates the first and second options as being invalid, and returns the third and fourth.

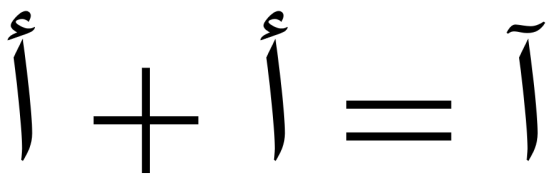

Figure 5: This diagram shows how two alifs with hamzas are combined into one alif with an elided hamza.

- `check_weak_postconjugate`
- `which_form`
- `check_root_filled`
- `check_root_hamza`
- `weak_in_root`

Check invalid before deconjugating. This function looks at the raw text to confirm that the word is longer than 2 characters. If the word is two characters before deconjugation, then it is impossible for it to be a correct verb. In this case, it is marked as invalid.

Check double hamza. This function looks at the first word to find an alif character with an elided hamza on the top. If there is an elision, it means there have been two hamzas combined. The function then splits the elided hamza into two separate alif hamza characters. This can be seen in Figure 5.

Deconjugate. This function begins by checking if the last letter is doubled with a shadda diacritic mark. If one is present, it marks the word as geminated and splits the doubled letter into two letters. It then continues with the deconjugation algorithm.

The deconjugation algorithm first identifies letters in the string based on their position: first, second, last, second-to-last, etc. If a word is not long enough to have a particular letter, that variable is marked as `None`.

The algorithm continues by following a series of logic statements to see whether the letters in particular positions of the word match the letters in a given conjugation pattern. For instance, the word *yf3lun* would go through the algorithm until it reaches the logic branch for words that begin with *y* and end with *un*, marking it as a verb that is in the present tense, third person, masculine, plural, and in the indicative mood. This feature set is added to the set of feature sets within the Word object.

The algorithm does not test every case in an iterative fashion. Rather, it tests one letter at a time until it reaches the end of a branch. In the case of *yf3lun*, it begins by checking that the first letter is *y*. Then it checks if the last letter is *n*. Finally it checks if the second-to-last letter is *u*. Each time it checks a letter, if that letter is confirmed to be there, it adds +1 to either the `prefix_count` or `suffix_count` depending on if the letter is at the beginning or end of the word. Note that these counters do *not* apply to the prefixes and suffixes previously identified by the `create_possible_words` function; these prefixes and suffixes are within the pattern of the conjugation only.

Stripping affixes. The algorithm then strips the prefixes and suffixes from the text string based on the prefixes and suffixes that were counted during the deconjugation step.

Checking weak post-deconjugation. The algorithm counts the remaining letters in the word after the conjugation affixes have been dropped to see if the word is at least three letters long. If it is only two letters long, that means that a vowel was dropped somewhere, which indicates that there is a vowel in the root and the word is weak. In this case, the `weak` Boolean is marked as True. If the verb is marked weak here, it is too short to be in any form besides Form I, so the form is set to Form I.

Determining the form. The `which_form` algorithm works fairly similarly to the `deconjugate` algorithm, in that it looks at the letters in a word one at a time to check if they match a particular verb form. There are also helper functions for each form.

The string that this algorithm compares its letters against is the deconjugated version of the word, with prefixes and suffixes dropped. In most cases, the deconjugated verb is the same as the past tense third person masculine singular conjugation, which is the "default" that verb forms are presented in (i.e, in Fig. 1, every word is written in the past tense third person masculine singular conjugation). In the cases of Form IV, Form VIII, and Form X, it is possible that the initial alif (the "a" letter) is dropped as a result of the conjugation; when the `which_form` function checks these forms, it looks at the deconjugated verb with and without the "a".

The `Word` object has a `checked_forms` set which stores all verb forms that have already been checked for that word. As the deconjugated verb goes through the logic tree, at each step it confirms that the form has not already been checked before then looking at the individual letters within the word. When a word has enough distinct letters and/or matches the length of a particular pattern in a given form, the word is run through a helper function for that form (e.g., `check_iii`) which checks every letter in the word against the letters in the form. If the form matches the word perfectly, the root is stored, which is identified by looking at the position of the root letters within that form.

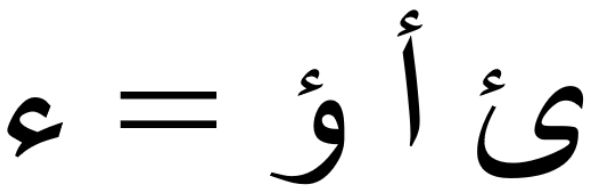

Figure 6: The various ways a hamza can appear in text.

If the word makes it through the entire `which_form` algorithm and does not match any of the forms, it is marked as invalid.

Checking that the root is filled. At this point in the algorithm, it is necessary to check that a root has been identified for the word. If no root has been assigned, the root defaults to the deconjugated version of the word, with no prefixes or suffixes. There are two reasons this is necessary. In the case of weak verbs, if a vowel is dropped the verb would not match any of the form patterns. Previously in the algorithm, in `check_weak_postconjugate`, the form would be marked as Form I in this case, but the root would not be assigned.

The second reason this is useful is in the case of very rare words with four-letter roots. These would not match any of the form patterns, but would still be deconjugated appropriately to produce their past tense third person masculine singular form. In these instances, it is important to show the user that the verb has been deconjugated and information could be gleaned from the conjugation pattern, but that the root has four letters, which are all the letters in the past tense third person masculine singular conjugation.

Checking for hamza in the root. Next, the algorithm looks for any vowels with a hamza in the root. A hamza is a glottal stop in Arabic, and when written, it is often written over vowels. So while it is possible to have a hamza in the root, they often do not appear as hamzas alone; they are an entirely different character on computers, so it is necessary to check for all possible hamza characters and convert them into standard hamzas. This is shown in Figure 6.

Checking for weak root. In the case of weak verbs, if the root vowel is not dropped, our algorithm will correctly identify the root vowel as a part of the root. However, weak verbs often change the written vowel depending on the unwritten short vowels and the pattern the verb is in. For example, if an "a" is written in a past tense hollow verb, it does not mean there is an "a" in the root; rather, it means there is an "o/u" or "y/i" in the root. In another example, if a past tense defective verb ends in an alif maqsura (a special type of "a"), it means that there is a "y/i" in the root; conversely, if a past tense defective verb ends in a "y/i", it means that there is an alif maqsura in the root.

Here, our algorithm checks for these discrepancies and corrects them.

5.6 Sanity Check

At this point in the algorithm, the function `pipeline` has completed, as all information that can be gathered about a verb has been collected and stored in the `Word` object. Before returning the list of words and their associated information, it is necessary to confirm that none of the information stored in these words is conflicting with itself.

There are four steps to our `sanity_check` function, as outlined below:

- `shadda_in_root`
- `check_hollow_past`
- `check_future`
- `check_prefix_order`

Checking for shadda in root. The shadda is the Arabic diacritic denoting that a letter is doubled and therefore pronounced in a stronger way. If a verb goes through our algorithm and determines that there is a shadda in the root, it means that this cannot be correct information about the word. In these cases, the word is marked as invalid, so it can be removed from the final word list.

Checking feature sets for hollow verbs in the past tense. Typically, there are four feature sets that go together in the past tense, because they are all ambiguous, ending in the same letter "t":

- Past tense first person singular (no gender) = "I did"
- Past tense second person masculine singular = "you (m) did"
- Past tense second person feminine singular = "you (f) did"
- Past tense third person feminine singular = "she did"

However, in the case of hollow verbs, these are not all ambiguous. If a verb is hollow, then the vowel may appear as an "a" in the past tense, in which case it is possible to distinguish between these sets. If the "a" is written, then the only feature set that applies is the past tense third person feminine singular ("she did"). If the "a" is not written, then this feature set does not apply at all.

The `check_hollow_past` function looks for exactly this. If the verb is hollow, the last letter is "t", and the "a" is present, then the other feature sets are discarded from the set of feature sets. If the "a" is not present, it discards the past tense third person feminine singular feature set and keeps the rest.

Checking future tense. In Arabic, the future tense is marked with the "s" prefix attached to a verb conjugated in the present tense. The `check_future` function confirms that if a verb has the "s" prefix, then it is marked in the future tense and it must otherwise be conjugated in the present tense. If a verb has the "s" prefix and is conjugated in the past, then it is marked as invalid.

Checking prefix order. The last thing our algorithm does to confirm that a word is correct is to check that if any prefixes were dropped, they were originally attached in the correct order. In this context, "prefix" refers to the prefixes being iterated through as part of the `create_possible_words` function; this does not include the "prefixes" dropped as part of deconjugating the verb.

In Arabic, it is necessary that these prefixes are in a specific order, or else they are not grammatically correct. It is the difference between saying "and I will go" compared to "will and I go", where the latter is nonsensical to an English speaker. If our algorithm drops prefixes and they turn out to be in an order that is grammatically incorrect, it means one of two things: either the user entered a word that is not grammatically correct, or the algorithm counted a letter as a prefix that was not actually a prefix. This is why it is necessary

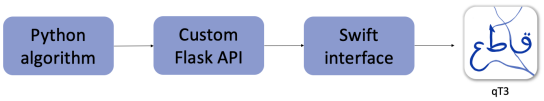

Figure 7: A diagram showing the overall flow of technologies leading to the functioning app, represented by our app icon.

to confirm that the prefixes are in the correct order, because if it is a result of the algorithm, it is important to eliminate the word with the invalid prefix as a possibility before returning the final information to the user.

Our algorithm is able to accomplish this confirmation because the dropped prefixes were originally stored in a list which maintains order. The `check_prefix_order` function first looks to see if there are 2 or more prefixes in the `dropped_prefix` list of the `Word` object, and then iterates through them.

The global list of prefixes, `verb_prefixes`, contains more information than just the Arabic text and English translation. Each element in this set also has an integer value associated with where it may appear in the order of prefixes; these values are hard-coded because they are specific to how Arabic grammar dictates prefixes must be ordered. For example, the "s" (future) prefix must always appear last, so it has the largest integer value of all the prefix elements. The "w" prefix, meaning "and", must always appear first, so it has an integer value of zero. The "f" prefix, meaning "then", also appears first and cannot appear simultaneously with the "w"/"and" prefix, so it also has an integer value of zero.

Using these integer values, the `check_prefix_order` function compares each prefix to the prefix following it to confirm that the integer value of the second prefix is larger than the first. Since verbs can have up to 3 combined prefixes, it was necessary to do this in a `for` loop rather than simply comparing two values.

5.7 Returning the words

At this point in the algorithm, a complete list of possible words has been generated in response to what the user initially entered. The `Word` objects have been filled with information, and if necessary, marked as invalid.

Before returning the list of words to the API, the possible words are removed from the list if they are invalid, and the list length is checked to confirm that it has at least one word. If there are no words in the list, an empty `Word` object with form "None" is appended so the user can see that no proper verbs were found.

5.8 Connection of algorithm to app

We created a custom API with Flask to be able to deploy and access the Python algorithm. The API was deployed through Heroku. This pipeline of technologies is shown in Figure 7. From there, our app was able to make calls to the API based on the verb that was entered by the user. The design of the API was simple but essential: a `GET` call takes in a verb and runs it through the entire pipeline. A JSON object is then returned which includes all of the essential information about the word. From there, the app is able to return this information on the front-end.

6 RESULTS

6.1 iOS Application

Our primary deliverable on this project was a functional iOS app. The app takes a verb from the user as shown in Figure 8. If the verb is valid and deductible by our algorithm, the app will return an informational page like the ones shown in Figures 9 and 10. If the verb input is gibberish or not deductible by the algorithm, the app will return the form as "None" and leave the rest of the information blank. Similarly, if the app detects English characters, the app will simply return the form as "English." These features are displayed in Figure 11.

One of the most important considerations for designing the app was how to deal with the inevitable ambiguity. One case for ambiguity is ambiguity of form, since some forms can overlap in certain situations, in which case both returned forms are displayed as shown in Figure 12. Another important case was the ambiguity in the word caused by potential prefixes or suffixes. In these cases, the app provides two links, with each one linking to a separate page of information with that word. The user is then able to validate which one is appropriate or a real word by clicking on the word, which links to the word's Wiktionary entry if one exists. Additionally, each entry will retain the information about the prefix or suffix that was dropped so that the user can view and use any context clues they might have to validate or invalidate the result. This information is shown in a popup as shown in Figure 13. The informational tag displayed is also shown in Figure 10.

6.2 Testing Suite

We also developed a testing suite for our algorithm to be able to ensure its functionality across different cases. The testing suite included 30 separate tests, some of which have several tests within, and they all pass. While there are certainly more cases not encompassed by this suite, the testing suite is comprehensive to our knowledge. Some tests focused on different parts of the pipeline in order to get a fuller coverage of testing and to understand and pinpoint any potential problems. Others test particular inputs such as weak and defective verbs to test them through the entire pipeline and ensure that the resulting information is correct.

Figure 8: Input screen

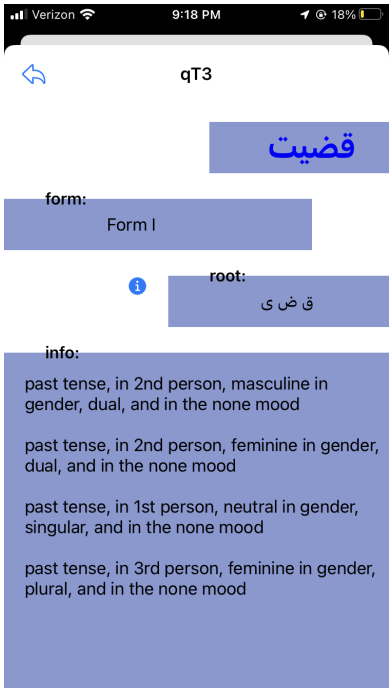

Figure 9: The informational page that the app returns given this input, which is an example of a defective verb. Note the informational flag which informs the user that this is a weak verb.

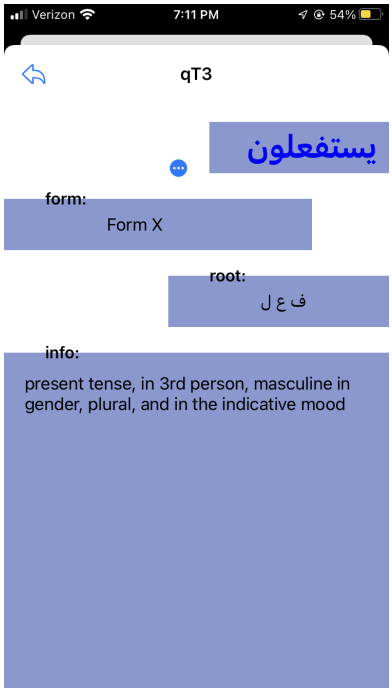

Figure 10: The information returned by the app on a regular Form X verb. This image also shows the "more info" tag which, if clicked, will display information about the input prefix(es) that were dropped. The pop-up from this tag is displayed in Figure 13.

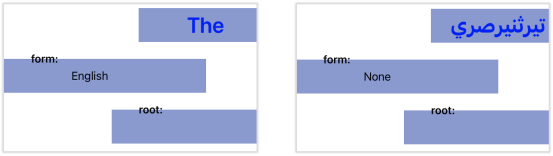

Figure 11: These screenshots show the results displayed if the algorithm detects English or is not able to detect a form, respectively.

Figure 12: Forms I and IV are often ambiguous with each other, in which case the resulting form is displayed like this. This ambiguity is also possible with Forms II and V.

7 DISCUSSION

In summary, we produced a functional iOS app that works on most Arabic verbs to be able to deduce the verb's form, root, and features—tense, gender, person, mood, and number. Our app fills a gap in previous resources available to Arabic students: it allows students to gain information about a verb that will help lead them to meaning and provide greater understanding than a simple lookup or translation app. Furthermore, it is more accessible to students than looking up a word in an Arabic dictionary, which requires already knowing the root of the word. The app also deduces all this information algorithmically, essentially breaking down a natural language in a logical and systematic manner through code.

7.1 Takeaways

One of the key takeaways of this work and something we were extremely conscious of as relevant to our objective was the ambiguity of certain information. We decided early on that we would not require short vowels to be input since they are rarely written in examples of Arabic text, and therefore users might not know the short vowels of their input word. This decision brought on many examples of ambiguity within the language itself. We had to make many decisions about how to deal with this ambiguity, both from an algorithmic standpoint and in terms of how to display this information to the user.

7.2 Limitations

The app was somewhat limited by the scope of the project. Since we did not have any database of valid Arabic words from which to draw, we were not able to independently validate user entries or words deduced by removing prefixes or suffixes. We link words to their

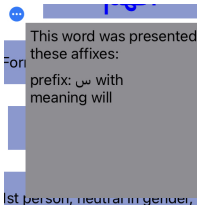

Figure 13: The popup displayed from the suffix/prefix "more info" tag.

Wiktionary entries, but this is limited by the crowd-sourced nature of Wiktionary, and is certainly not comprehensive nor necessarily correct. Additionally, there are likely a few bugs, and Arabic cases not tested by our test suite that our app may not work correctly for.

7.3 Future work

Our app could be improved for user experience by increasing the information returned to the user to account for students who are extreme beginners in Arabic. An extension of this app could include an informational page about how forms and roots function, as well as a guide for the user on how to use the app. Creating a mouse-over that describes the typical meaning of each form would also help more novice users as well.

Another possible extension to our project would be to expand the testing suite to be more rigorous and get a fuller picture of where our code may fail.

8 ACKNOWLEDGMENTS

We would like to especially thank Professor Daniel Scharstein and Ustaaz Usama Soltan for their assistance in the making of this project. We would also like to thank Professor Pete Johnson and Professor Christopher Andrews for their support.

REFERENCES

[1] H. Al-Omari. Almas: An arabic language morphological analyzer system. 1994.
[2] Arabic Desert Sky. Arabic desert sky: Arabic learning resources, 2007. [Online; accessed 16-December-2021].
[3] T. Buckwalter. Qamus: Arabic lexicography, 2002. [Online; accessed 16-December-2021].
[4] A. Castaño. Multimedia english: Assimilations of the /j/, 2008. [Online; accessed 16-December-2021].
[5] Elementary Arabic. Elementary arabic: Introduction to verb measures, 2012. [Online; accessed 16-December-2021].
[6] T. Sembok, B. Abuata, and Z. Bakar. A rule and template based stemming algorithm for arabic language. *International Journal of Mathematical Models and Methods in Applied Sciences*, 5, 05 2011.
[7] Wikipedia contributors. Classical arabic — Wikipedia, the free encyclopedia, 2021. [Online; accessed 16-December-2021].
[8] Wiktionary contributors. Arabic verbs — Wiktionary, the free dictionary, 2021. [Online; accessed 16-December-2021].