INTRODUCTION

While I was tackling a NLP (Natural Language Processing) problem for one of my project "Stephanie", an open-source platform imitating a voice-controlled virtual assistant, it required a specific algorithm to observe a sentence and allocate some 'meaning' to it, which then I created using some neat tricks and few principles such as sub filtering, string metric and maximum weight matching.

ALGORITHM

The algorithm uses Levenshtein Edit Distance and Munkres Assignment Algorithm to tackle problems like string metric and maximum weight matching. Let's get an overview on each of the given algorithm as described above:

LEVENSTEIN EDIT DISTANCE

In information theory and computer science, the Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. It is named after Vladimir Levenshtein, who considered this distance in 1965.

Levenshtein distance may also be referred to as edit distance, although that term may also denote a larger family of distance metrics. It is closely related to pair wise string alignments.

In layman terms, The Levenshtein algorithm (also called Edit-Distance) calculates the least number of edit operations that are necessary to modify one string to obtain another string.

```
l e  v e n s h t e i n        l e v  e n s h t e i n
o = + o = = = - = = = =  or   o = o + = = = - = = = =
m e i l e n s  t e i n        m e i l e n s  t e i n
```

"=" Match; "o" Substitution; "+" Insertion; "-" Deletion

It is been used to calculate the score which denotes the chances that given word is equal to some other word.

MUNKRES ALGORITHM

The **Hungarian method** is a combinatorial optimization algorithm that solves the assignment problem in polynomial time and which anticipated later primal-dual methods. It was developed and published in 1955 by Harold Kuhn.

James Munkres reviewed the algorithm in 1957 and observed that it is (strongly) polynomial. Since then the algorithm has been known also as the **Kuhn–Munkres algorithm** or **Munkres assignment algorithm**.

Let's use an example to explain it even further, there are three workers: Armond, Francine, and Herbert. One of them has to clean the bathroom, another sweep the floors, and the third wash the windows, but they each demand different pay for the various tasks. The problem is to find the lowest-cost way to assign the jobs. The problem can be represented in a matrix of the costs of the workers doing the jobs. For example:

	Clean bathroom	**Sweep floors**	**Wash windows**
Armond	$2	$3	$3
Francine	$3	$2	$3
Herbert	$3	$3	$2

The Hungarian method, when applied to the above table, would give the minimum cost: this is $6, achieved by having Armond clean the bathroom, Francine sweep the floors, and Herbert wash the windows.

Similarly it can be used to compute the maximum cost by doing a small alteration to the cost matrix. The simplest way to do that is to subtract all elements from a large value.

METAPHONES

Metaphone is a phonetic algorithm, published by Lawrence Philips in 1990, for indexing words by their English pronunciation.

Similar to soundex metaphone creates the same key for similar sounding words. It's more accurate than soundex as it knows the basic rules of English pronunciation. The metaphone generated keys are of variable length.

The use of metaphone is to get a more generic form of a text especially names such as stephanie ends up as "STFN", stefanie also ends up as "STFN", while tiffany would end as "TFN".

They pretty much can be used in addition to current given algorithms in case the input data contains lot of generic information/unrevised information.

PUTTING IT ALL TOGETHER

Algorithm takes two parameters, dataset and query.

Dataset is a 2D list comprised of words, for a dummy example, this could be a dataset:

[['twitter', 'notifications'], ['unread', 'emails'], …]

And now query will be a list of given keywords that needs to be searched, for instance:

['give', 'twitter', notifs']

Now we will iterate through each row (list) of 2D list (dataset) and compare it to query list in this given way.

- Now we will have two single dimensional lists like [‘twitter’, ‘notifications’] and [‘give’, …] for the very first iteration of the dataset in the above given example.
- We use double loop to compare each word of the first list with each word in other list, like comparing ‘twitter’ with ‘give’, then ‘twitter’ with ‘twitter’…, then ‘notificaitons’ with ‘give’, ‘notifications’ with ‘twitter’ and so on.
- Comparison is done using Leinshtein Distance, this results in a value ranging from 0.0 – 1.0 where 1.0 equates to the fact that both of the words are same.
- Now we have a one to one comparison of each word present in both of the lists, now we can create a matrix out of it whose dimension will be size of first list x size of second list. In this given example, 2 x 3 which will have values computed from levenshtein embedded in it.
- Now this matrix can used to compute the maximum assignment using Munkes algorithm, which takes the given algorithm and returns back the maximum scores index.
- This returned matrix is a vector matrix with the size of number of rows of the cost matrix (matrix that was sent to munkes algorithm) denoting which values gives the maximum output.
- Now we initialize a list and create a loop, which goes from 0 to the length of data list (ith member of dataset) and give value to each word present in the data list computed from munkres algorithm which basically denotes that chances that each word of data is present in query, (for ex- what’s the chance that the word ‘twitter’ is present in the query, ‘notifications’ and so on)
- Now we get these this list and compute it’s average by summing up all the values (chances that given word is present) and divide it by the number of words present in it (length).
- Finally we calculate the maximum average of the entire dataset and return its index. (In case there are two indexes which has same average we choose the one which has higher sum (values)).

On a side note, metaphones can be used in addition to given concept where all of the dataset and query is first converted into respective metaphones before sending it to this master algorithm, increasing the efficiency of the algorithm quite drastically especially with generic names. ("Stephanie" = "STFN"), ("Tiffany" = "TFN") and is actively used in Stephanie, to check the availability, (“Stephanie,…”), or (“Hey Stephanie, wake up”), since it works best with small input data.

PRACTICAL APPLICATION

For the starters, "Stephanie" is a virtual assistant inspired by various other products in the market such as "Siri", "Cortana" and "Watson" to name a few. It's written in python and acts more like a framework of some sort than a core application, where 3rd party modules can be injected into it using a really simple developer API.

So how Stephanie "thinks" is that when a user speaks through an audio device to Stephanie, it records that voice and saves it as .wav file in the memory and sends the contents of that file to one of the leading voice recognition services provided by various services such as "Google cloud speech", "Bing speech API", "wit.ai" and various others after then it receives a response which is usually a string containing the "meaning" of the sentence based on the learning algorithm that service uses, so the end product is a text which is usually what speaker meant now the goal of Stephanie is to somehow use that text to learn what user meant and fulfill it's potential needs.

APPLICATION LOGIC

The application logic of Stephanie works somewhat this way:

i. The modules in Stephanie are defined as separate classes

ii. Each class inherits from a base class which handles all the dependencies and provides an interface layer to core functions of the application such as .speak("Good evening, Sir"), .listen(), .decipher(), .search(), etc methods, so that developers can interact with the main application quite easily by just calling the above methods.

iii. Each module defines a set of keywords which when found in the users "spoken" text (retrieved from one of the 3rd party services such as "Google speech" and so on.) instantiates that specific module and calls a method on it.

That method is also defined in the class, so now what the algorithm does is that it takes the full text as one of its input while the other input is all the keywords of different modules in some kind of array format, and then it results with the index which algorithm decides is the best guess of the given modules.

PSEUDO CODE

A really simple demo module in this case twitter module which gets some twitter notifications.

```
TwitterModule(BaseModule): // Base module handles all the dependencies.

    def __init__(self, *args):

        super().__init__(*args)

    def get_notifications(self):

        // use some logic to get notifications by hooking with Twitter API

        response = "You have %s notifications, namely %s" % (data['count'],
                                                             data ['text']
```

```
self.assistant.say(response)
```

TwitterModule has its keywords assigned as ['twitter', 'notifications'], though to be a little more specific and help developers writing rules it's written as:

```
["TwitterModule@GetNotifications", ['twitter', 'notifications']]
```

- "TwitterModule" is the classname.
- "GetNotifications" is the method used to handle that function, it's converted into snake_case and invoked dynamically.
- ['twitter', 'notifications'] are the keywords which co-relate Twittermodule to intended text.

IN PRACTICE

So now whenever a user speaks something to Stephanie, it takes the voice, gets the intended text in string format, it's then cleaned a little bit to get rid of irregularities and split into an array with delimiter parameter set as " ". This array is then sent to another text processing mechanism where all the "sub words" (words which hold somewhat lower precedence such as "a", "an", "the", "I", "you", "what", "which", "has", etc basically all the articles, tenses and some prepositions) are filtered into another array which for now is left unused.

So, now we have two arrays one with keywords spoken by a user and second is a 2D array with module information and keywords co-relating them, we take the keywords part of that 2D array and then pass user_keywords and module_keywords (it's still 2D array) to our algorithm which chunks the result in this case an index value which relates the element in that array (module) with which the user keywords are most identical with.

POTENTIAL OPTIMIZATION

- Sub words shouldn’t be filtered out completely instead can be filtered down to priority scales. For instance “a”, “an”, “the” could resonate to 1 as they don’t hold much importance to the meaning of sentence, while “I”, “You”, “We” could resonate to 2 since they are a bit more meaningful and so on.
- Deep Neural Networks can be used alongside this algorithm to predict the intent of a given sentence by using correct data sets and modeling.
- Actual code provided can be optimized with techniques like memorization, using numpy instead of built-in lists.

Conclusion

The working efficiency of this algorithm is pretty good, and I would highly recommend checking the code provided in the github to gain more of the hidden insight and see it work in practice. This search algorithm acts as a primary core of Stephanie which helps it determining the ‘meaning’ of the sentence and triggering the correct response. I highly recommend checking Stephanie as well since it’s open-source and you could see this phonetic algorithm in real life application.

P.S. I am 18 year old lad, who didn't go to college and most of my understanding of programming and computer science fundamentals came from active surfing, reading lots of books, and just plainly asking questions and finding it's answers. So I doubt this as more of a blog than a research paper, so kindly go easy on me and if you find any mistake or just want to improve the given document or just wanna talk about it in depth or merely for fun, just contact me at ugupta41@gmail.com.