Artificial Intelligence CS3642
Semester Project
Fall 2023
Dr. Hung
Kennesaw State University
Faero.ai

Research

Phonetic Partitioning File Audio Transcription

Rene Lisasi

College of Computing and Software Engineering
rlisasi@students.kennesaw.edu

Abstract: Speech to text is widely available and nothing new. The available tools to complete speech transcription from direct input (like a microphone) are good but fall short during audio file transcription (mp3/wav). The aim of this paper is to research and implement a novel audio transcription method that is hopefully more accurate, lightweight, faster, and requires less training than conventional speech to text models like CMU-Sphinx or Google's Speech Recognition. This novel method employs cosine similarity to break down words into phonetic partitions that are compared to a lightweight dataset of formant vectors that correspond to a sound which has possible syntactic outcomes (letters). The result of this research is a relatively simpler, novel way to transcribe audio specifically designed to tackle file audio.

Keywords: formant; phonetic partition; cosine similarity; vector;

1. Introduction

In the realm of audio transcription, the implementation discussed herein represents a notable stride towards efficiency and speed. The current version is proficient in transcribing audio but stands as a work in progress. The accuracy achieved is on par if not better than that of CMU-Sphinx, highlighting the innovation embedded in this approach and lays a robust foundation for exploration and research.

As we delve into the conclusions drawn from this study, the core objective was to devise a novel and straightforward way to transcribe audio from files. Traditional technologies are often tailored for direct speech or are commercialized software, leaving a gap in addressing the broader spectrum of audio data. Here, my focus is on utilizing cosine similarity to compare audio file segments and hopefully open new avenues for areas unlike music and acoustics which drift away critical junction of human data recognition. It is imperative to redirect our attention towards the development of technologies catering to the nuances of human communication like human to robot interactions. The aim is to streamline and enhance organic human data recognition and diminish the reliance on peripheral communication mechanisms and bridge the gap between human expression and self-digitization.

2. Linguistics

2.1. Formants

To implement audio transcription in a more efficient way, one must understand linguistics. The core principle of linguistics is that formant values determine speech. Formants are made up of energy waves emitted from someone's mouth as they speak. because humans have oval tracts, the shape of the waves have unique features. Those features are our formants. They are 3 superimposed waves of high energy, the lowest energy formant f1, is the primary determinant of vowels, while f3, the one with the highest frequency determines consonant sounds. Consonant sounds typically have lower energy levels than vowels due to the tract shape we make when pronouncing them. With that being said, we will analyze the make-up of a word using a vector of these 3 formant values. How? you might ask, The details are up next.

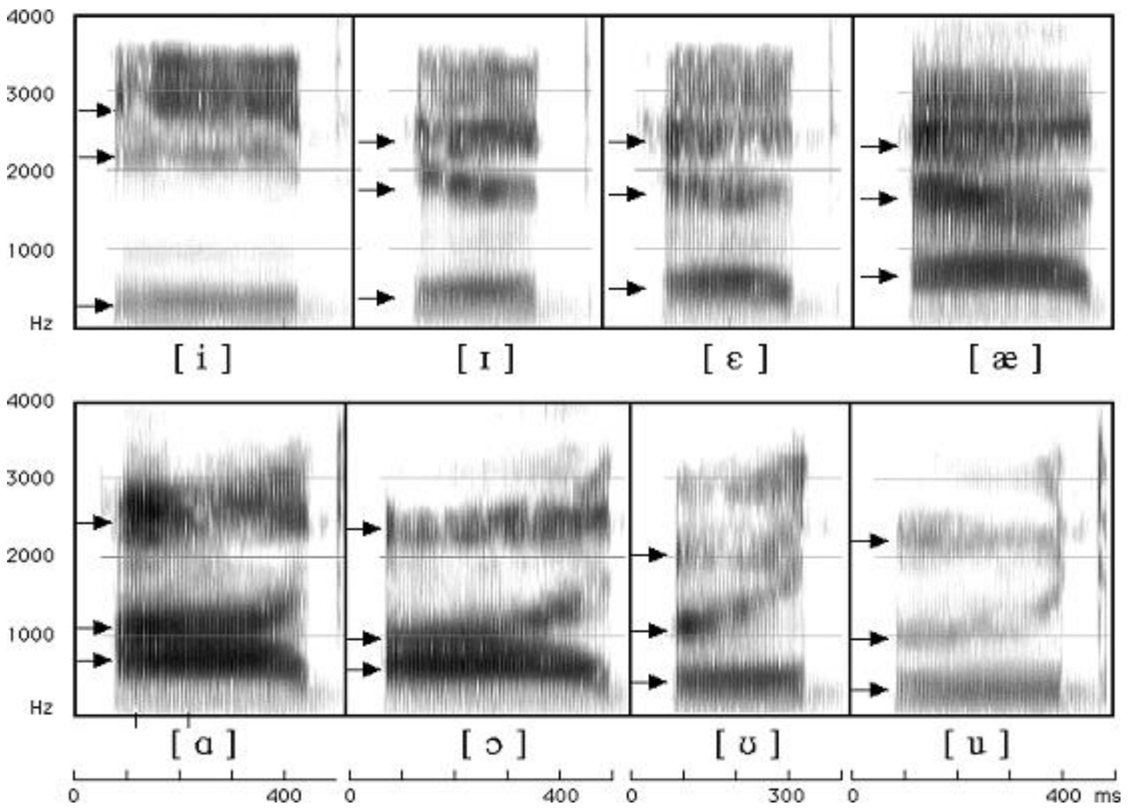


Figure 1. Example of vowel sounds and their formants

In the above image:

- The bottom arrows are f1 values, middle arrows are f2, and top arrows are f3 values.
- It's best to analyze formants in terms of frequency (Hz) and time (ms).
- The darkest areas of the spectrogram are the formant peaks.
- Following the formant peaks of one formant gives us a formant equation.
- Ideally, we'd like to compare the equation of and input frame to that of a known frame, but in this implementation we compare the averages, how can we do this?
- There are many ways to analyze and compare the formant values but the way I chose to do it was mean summation.

Mean summation: We sum up the highest peaks and divide them by the total number of highest peaks like this:

$$f_{1_{average}} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

$$f_{2_{average}} = \frac{1}{m}\sum_{j=1}^{m} Y_j$$

$$f_{3_{average}} = \frac{1}{p}\sum_{k=1}^{p} Z_k$$

- $f_{1_{average}}$ is the mean of all f1 values for a given audio slice.
- $f_{2_{average}}$ is the mean of all f2 values for a given audio slice.
- $f_{3_{average}}$ is the mean of all f3 values for a given audio slice.

We do this for all vowels and consonants on the IPA vowel and consonant charts respectively. The problem is there are over a hundred consonants and many possible letters that could be a result of those sounds. IPA charts classify sounds like this for example: Plosive Bilabial which corresponds to the sound /p/. It's hard to keep track of so many sounds without being a full linguist. Therefore another naming convention was used to keep track of sound names called "*x-sampa*" where the same /p/ sound has the name *p* and the /m/ sound has the name *m_0.* This will come in handy later.

These formant average values are the most fundamental part of phonetic partition transcription because they are the vectors from which a sound will be compared to another to determine if its children are the most likely letter being pronounced.

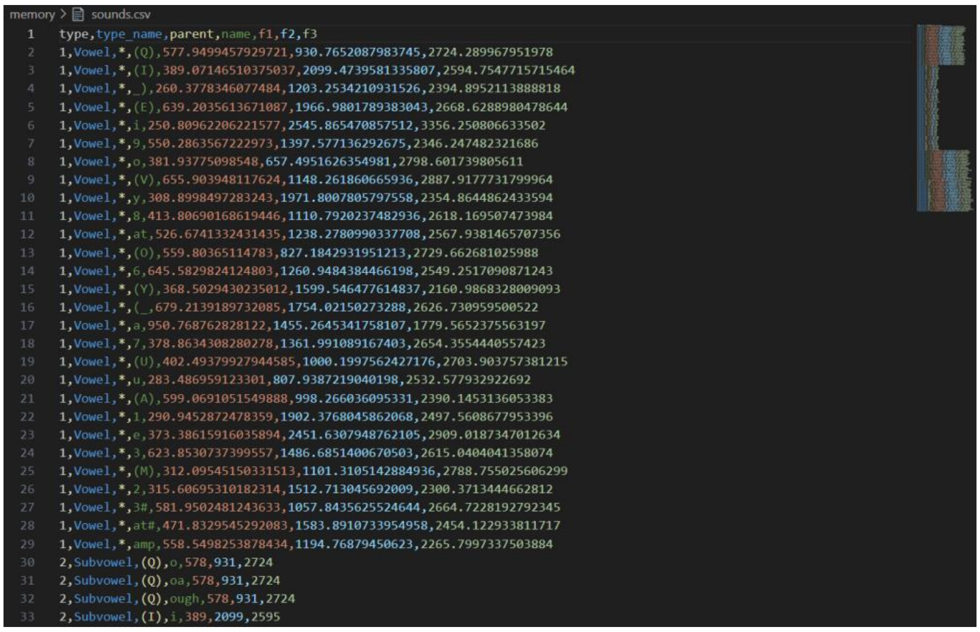
3. Implementation

3.1. Characteristics of the dataset

Now that we know what formant values are and how to collect them we will need a place to store them to be retrieved later for comparison with our input sound.

Each formant average is now a field of either a vowel, sub-vowel, consonant, or sub-consonant. The sub types are the children of the sound, but we will go into that structure later as it forms part of the comparison.

For now, we save the formants, type of sound, and parent as a record in a csv file, from here we will conduct a mass harvest of all the sounds at once using a signal processing library called Praat. You can use the signal processing library of your choice, but Praat is the fastest as it is built entirely in C++.



memory > sounds.csv

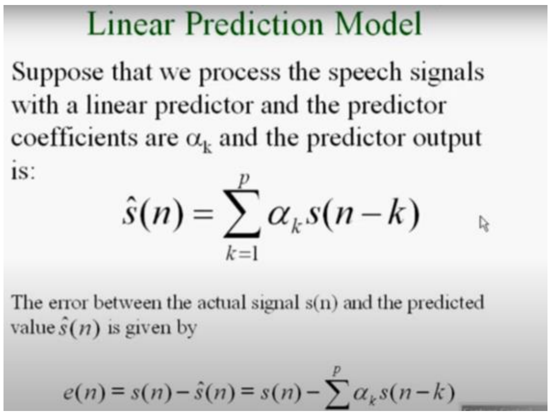
```
type,type_name,parent,name,f1,f2,f3
1,Vowel,*,(Q),577.9499457929721,930.7652087983745,2724.289967951978
1,Vowel,*,(I),389.07146510375037,2099.4739581335807,2594.7547715715464
1,Vowel,*,_),260.3778346077484,1203.2534210931526,2394.8952113888818
1,Vowel,*,(E),639.2035613671087,1966.9801789383043,2668.6288980478644
1,Vowel,*,i,250.80962206221577,2545.865470857512,3356.250806633502
1,Vowel,*,9,550.2863567222973,1397.577136292675,2346.247482321686
1,Vowel,*,o,381.93775098548,657.4951626354981,2798.601739805611
1,Vowel,*,(V),655.903948117624,1148.261860665936,2887.9177731799964
1,Vowel,*,y,308.8998497283243,1971.8007805797558,2354.8644862433594
1,Vowel,*,8,413.80690168619446,1110.7920237482936,2618.169507473984
1,Vowel,*,at,526.6741332431435,1238.2780990337708,2567.9381465707356
1,Vowel,*,(O),559.80365114783,827.1842931951213,2729.662681025988
1,Vowel,*,6,645.5829824124803,1260.9484384466198,2549.2517090871243
1,Vowel,*,(Y),368.5029430235012,1599.546477614837,2160.9868328009093
1,Vowel,*,(_,679.2139189732085,1754.02150273288,2626.730959500522
1,Vowel,*,a,950.768762828122,1455.2645341758107,1779.5652375563197
1,Vowel,*,7,378.8634308280278,1361.991089167403,2654.3554440557423
1,Vowel,*,(U),402.49379927944585,1000.1997562427176,2703.903757381215
1,Vowel,*,u,283.486959123301,807.9387219040198,2532.577932922692
1,Vowel,*,(A),599.0691051549888,998.266036095331,2390.1453136053383
1,Vowel,*,1,290.9452872478359,1902.3768045862068,2497.5608677953396
1,Vowel,*,e,373.38615916035894,2451.6307948762105,2909.0187347012634
1,Vowel,*,3,623.853073739957,1486.6851400670503,2615.0404041358074
1,Vowel,*,(M),312.09545150331513,1101.3105142884936,2788.755025606299
1,Vowel,*,2,315.60695310182314,1512.713045692009,2300.371344662812
1,Vowel,*,3#,581.9502481243633,1057.8435625524644,2664.7228192792345
1,Vowel,*,at#,471.8329545292083,1583.8910733954958,2454.122933811717
1,Vowel,*,amp,558.5498253878434,1194.76879450623,2265.7997337503884
2,Subvowel,(Q),o,578,931,2724
2,Subvowel,(Q),oa,578,931,2724
2,Subvowel,(Q),ough,578,931,2724
2,Subvowel,(I),i,389,2099,2595
```

Figure 2. CSV Dataset of each sound

3.2. Signal Processing

The reason we do not build our own is because there is no need to reinvent the wheel and signal processing is only a small part of transcribing speech to text. We'll be using Praat library to handle the signal processing.

Praat uses Linear Prediction Model (LPC) to predict the values that correspond to the matrix of values from the input signal:



Linear Prediction Model

Suppose that we process the speech signals with a linear predictor and the predictor coefficients are α_k and the predictor output is:

$$\hat{s}(n) = \sum_{k=1}^{p} \alpha_k s(n-k)$$

The error between the actual signal s(n) and the predicted value $\hat{s}(n)$ is given by

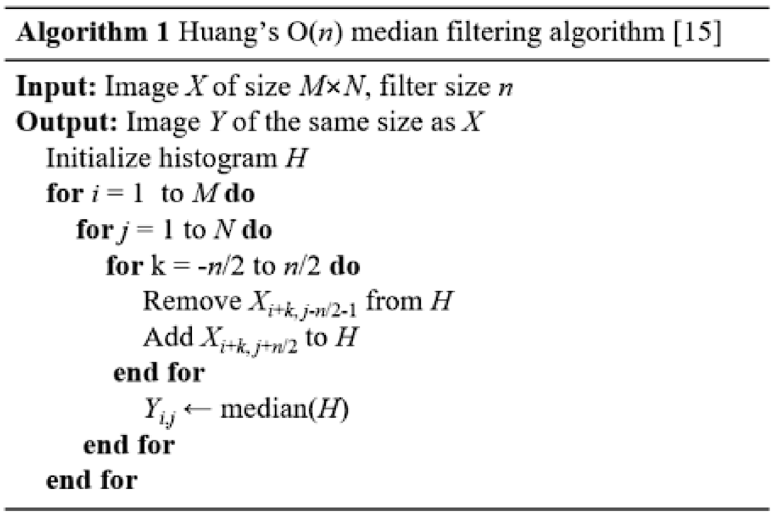
$$e(n) = s(n) - \hat{s}(n) = s(n) - \sum^{p} \alpha_k s(n-k)$$

Figure 3. Linear Prediction Model

If you decide you want to build your own signal processor as well, however, a common approach is to use Fast Fournier Transform among other techniques like LPC. One of the more interesting techniques that can be applied to signal process the sound into formant values is an image processing algorithm called median filtering. Median filtering is used to find the median value of the surrounding pixels given a starting pixel. One could alter this algorithm to find the max pixel instead of the median.

Making a list of max pixels would be the list of formant values from which one could collect the average and be right along with us.

Here is the pseudocode:



```
Algorithm 1 Huang's O(n) median filtering algorithm [15]
Input: Image $X$ of size $M \times N$, filter size $n$
Output: Image $Y$ of the same size as $X$
  Initialize histogram $H$
  for $i = 1$ to $M$ do
    for $j = 1$ to $N$ do
      for k = $-n/2$ to $n/2$ do
        Remove $X_{i+k, j-n/2-1}$ from $H$
        Add $X_{i+k, j+n/2}$ to $H$
      end for
        $Y_{i,j} \leftarrow$ median($H$)
    end for
  end for
```

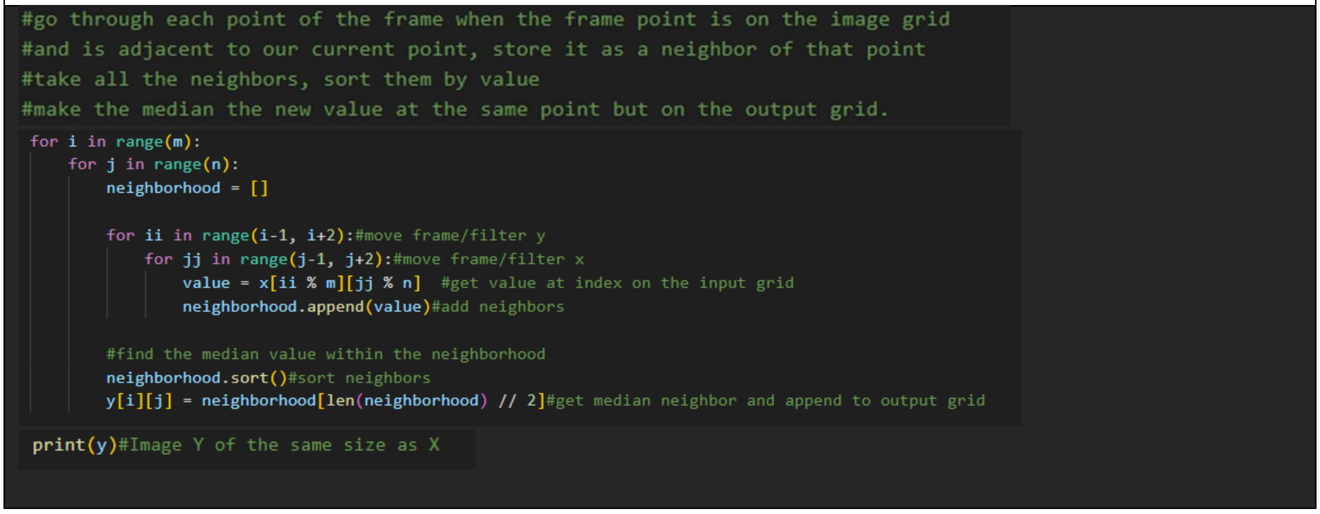
Figure 4. Median Filtering algorithm

Definition: *Median filtering is a nonlinear filter used to remove salt or pepper noise of an image or signal. Since the median filter is at strength for eliminating noise while preserving edges, it is a well-known algorithm in image processing .*

All you have to do is instead of collecting the median, after sorting the list of neighbors, collect the last index (largest value).

Here is the implementation for median filtering:

median_filter.py



```
#go through each point of the frame when the frame point is on the image grid
#and is adjacent to our current point, store it as a neighbor of that point
#take all the neighbors, sort them by value
#make the median the new value at the same point but on the output grid.
for i in range(m):
    for j in range(n):
        neighborhood = []

        for ii in range(i-1, i+2):#move frame/filter y
            for jj in range(j-1, j+2):#move frame/filter x
                value = x[ii % m][jj % n]  #get value at index on the input grid
                neighborhood.append(value)#add neighbors

        #find the median value within the neighborhood
        neighborhood.sort()#sort neighbors
        y[i][j] = neighborhood[len(neighborhood) // 2]#get median neighbor and append to output grid

print(y)#Image Y of the same size as X
```

Figure 5. Code for median filtering

3.3. Harvesting Process

The reason we must build this dataset is because there is a lack of single sound datasets available at the time of writing and could take up to 50% of the project. You must collect or record a sample of each possible sound on both IPA charts and implement the code to harvest the formant values for each of the sounds.

Along the way you must trim the sound to only the core of the sound, meaning no noise and no dead space.

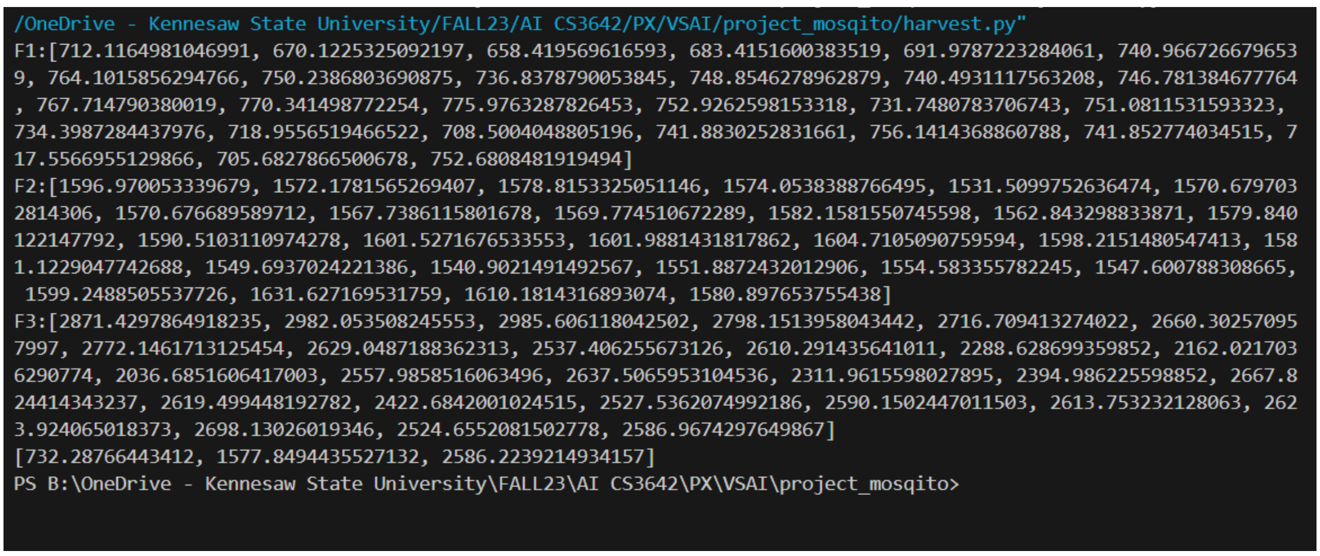
```
!pip install -U praat-parselmouth
```

harvest.py

```
import parselmouth
from parselmouth import praat
testfile = '/home/felix/data/data/audio/testsatz.wav'
sound = parselmouth.Sound(testfile)
f0min=75
f0max=300
pointProcess = praat.call(sound, "To PointProcess (periodic, cc)",
f0min, f0max)
formants = praat.call(sound, "To Formant (burg)", 0.0025, 5, 5000,
0.025, 50)
numPoints = praat.call(pointProcess, "Get number of points")
f1_list = []
f2_list = []
f3_list = []
for point in range(0, numPoints):
    point += 1
    t = praat.call(pointProcess, "Get time from index", point)
    f1 = praat.call(formants, "Get value at time", 1, t, 'Hertz',
'Linear')
    f2 = praat.call(formants, "Get value at time", 2, t, 'Hertz',
'Linear')
    f3 = praat.call(formants, "Get value at time", 3, t, 'Hertz',
'Linear')
    f1_list.append(f1)
    f2_list.append(f2)
    f3_list.append(f3)
```

Figure 6. Praat code

Output 1. Harvest output

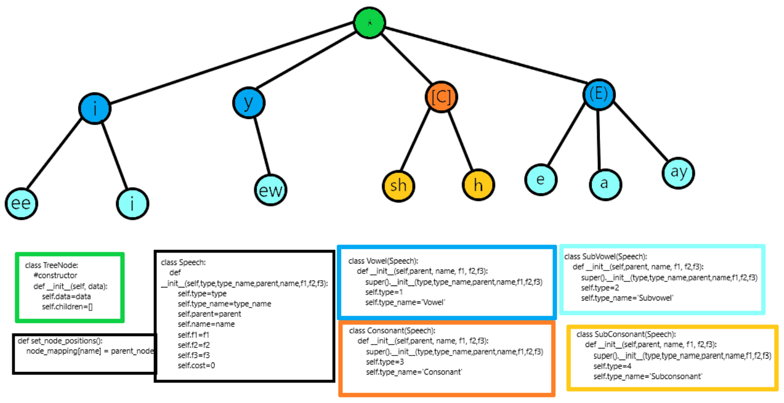


```
/OneDrive - Kennesaw State University/FALL23/AI CS3642/PX/VSAI/project_mosqito/harvest.py"
F1:[712.1164981046991, 670.1225325092197, 658.419569616593, 683.4151600383519, 691.9787223284061, 740.966726679653
9, 764.1015856294766, 750.2386803690875, 736.8378790053845, 748.8546278962879, 740.4931117563208, 746.781384677764
, 767.714790380019, 770.341498772254, 775.9763287826453, 752.9262598153318, 731.7480783706743, 751.0811531593323,
734.3987284437976, 718.9556519466522, 708.5004048805196, 741.8830252831661, 756.1414368860788, 741.852774034515, 7
17.5566955129866, 705.6827866500678, 752.6808481919494]
F2:[1596.970053339679, 1572.1781565269407, 1578.8153325051146, 1574.0538388766495, 1531.5099752636474, 1570.679703
2814306, 1570.676689589712, 1567.7386115801678, 1569.774510672289, 1582.1581550745598, 1562.843298833871, 1579.840
122147792, 1590.5103110974278, 1601.5271676533553, 1601.9881431817862, 1604.7105090759594, 1598.2151480547413, 158
1.1229047742688, 1549.6937024221386, 1540.9021491492567, 1551.8872432012906, 1554.583355782245, 1547.600788308665,
 1599.2488505537726, 1631.627169531759, 1610.1814316893074, 1580.897653755438]
F3:[2871.4297864918235, 2982.053508245553, 2985.606118042502, 2798.1513958043442, 2716.709413274022, 2660.30257095
7997, 2772.1461713125454, 2629.0487188362313, 2537.406255673126, 2610.291435641011, 2288.628699359852, 2162.021703
6290774, 2036.6851606417003, 2557.9858516063496, 2637.5065953104536, 2311.9615598027895, 2394.986225598852, 2667.8
24414343237, 2619.499448192782, 2422.6842001024515, 2527.5362074992186, 2590.1502447011503, 2613.753232128063, 262
3.924065018373, 2698.13026019346, 2524.6552081502778, 2586.9674297649867]
[732.28766443412, 1577.8494435527132, 2586.2239214934157]
PS B:\OneDrive - Kennesaw State University\FALL23\AI CS3642\PX\VSAI\project_mosqito>
```

To automate the harvesting process, modify your implementation to harvest entire directories of trimmed audio samples (one for each sound) at once and you're done harvesting.

3.4. Tree Structure

With the data being set, you want to set up a tree structure that has a maximum depth of two. This tree should have another class of Speech with subclasses vowels, sub-vowels, consonants, sub-consonants, and it should query the dataset into this tree at the start of the application.



```
class TreeNode:
    #constructor
    def __init__(self, data):
        self.data=data
        self.children=[]
```

```
def set_node_positions():
    node_mapping[name] = parent_node
```

```
class Speech:
    def
__init__(self,type,type_name,parent,name,f1,f2,f3):
        self.type=type
        self.type_name=type_name
        self.parent=parent
        self.name=name
        self.f1=f1
        self.f2=f2
        self.f3=f3
        self.cost=0
```

```
class Vowel(Speech):
    def __init__(self,parent, name, f1, f2,f3):
        super().__init__(type,type_name,parent,name,f1,f2,f3)
        self.type=1
        self.type_name='Vowel'
```

```
class SubVowel(Speech):
    def __init__(self,parent, name, f1, f2,f3):
        super().__init__(type,type_name,parent,name,f1,f2,f3)
        self.type=2
        self.type_name='Subvowel'
```

```
class Consonant(Speech):
    def __init__(self,parent, name, f1, f2,f3):
        super().__init__(type,type_name,parent,name,f1,f2,f3)
        self.type=3
        self.type_name='Consonant'
```

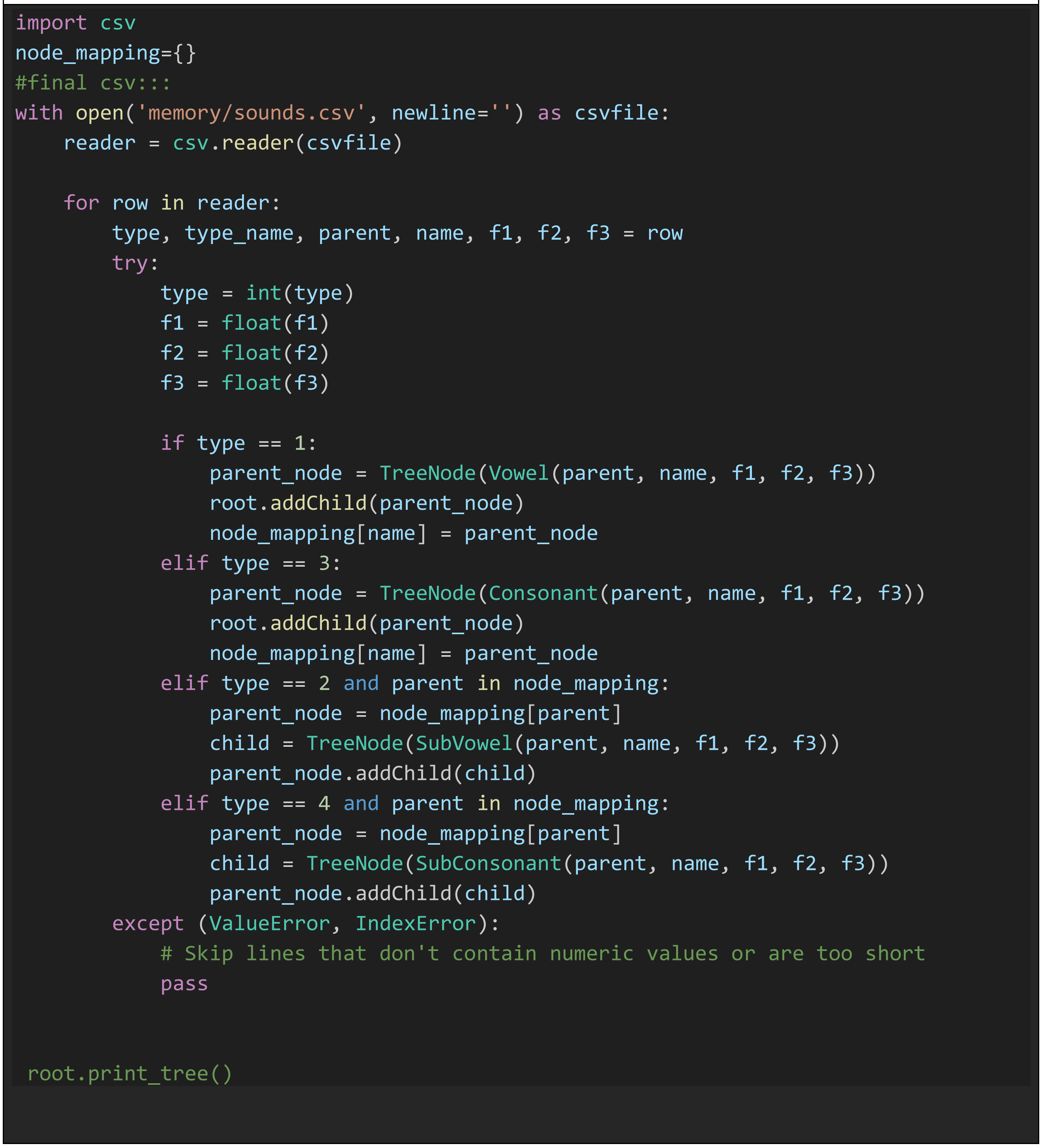
```
class SubConsonant(Speech):
    def __init__(self,parent, name, f1, f2,f3):
        super().__init__(type,type_name,parent,name,f1,f2,f3)
        self.type=4
        self.type_name='Subconsonant'
```

Figure 7. Tree structure for all sounds

3.4.1. Node mapping

The most important thing about this structure is how map links the children back to their parents so it is important to map the parents along the way then in the same function, add all the children to the right parents.

Part of binary_search_tree.py



```python
import csv
node_mapping={}
#final csv:::
with open('memory/sounds.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)

    for row in reader:
        type, type_name, parent, name, f1, f2, f3 = row
        try:
            type = int(type)
            f1 = float(f1)
            f2 = float(f2)
            f3 = float(f3)

            if type == 1:
                parent_node = TreeNode(Vowel(parent, name, f1, f2, f3))
                root.addChild(parent_node)
                node_mapping[name] = parent_node
            elif type == 3:
                parent_node = TreeNode(Consonant(parent, name, f1, f2, f3))
                root.addChild(parent_node)
                node_mapping[name] = parent_node
            elif type == 2 and parent in node_mapping:
                parent_node = node_mapping[parent]
                child = TreeNode(SubVowel(parent, name, f1, f2, f3))
                parent_node.addChild(child)
            elif type == 4 and parent in node_mapping:
                parent_node = node_mapping[parent]
                child = TreeNode(SubConsonant(parent, name, f1, f2, f3))
                parent_node.addChild(child)
        except (ValueError, IndexError):
            # Skip lines that don't contain numeric values or are too short
            pass


 root.print_tree()
```

Figure 8. Code for tree structure

3.5 Comparison

For the fun part we will compare a test input slice to one of the many 1st depth nodes of the tree. Once the list is sorted according to cosine distance, we choose the node with the smallest distance and then repeat the process for the children. The child with the smallest distance is our guess. Once we do that, we can slice a word into chunks the complete multiple comparisons to our tree to obtain the transcription of the audio.

What is cosine similarity or cosine distance?

3.5.1 Cosine similarity and distance

"Cosine similarity is a metric used to determine how similar two entities are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space."

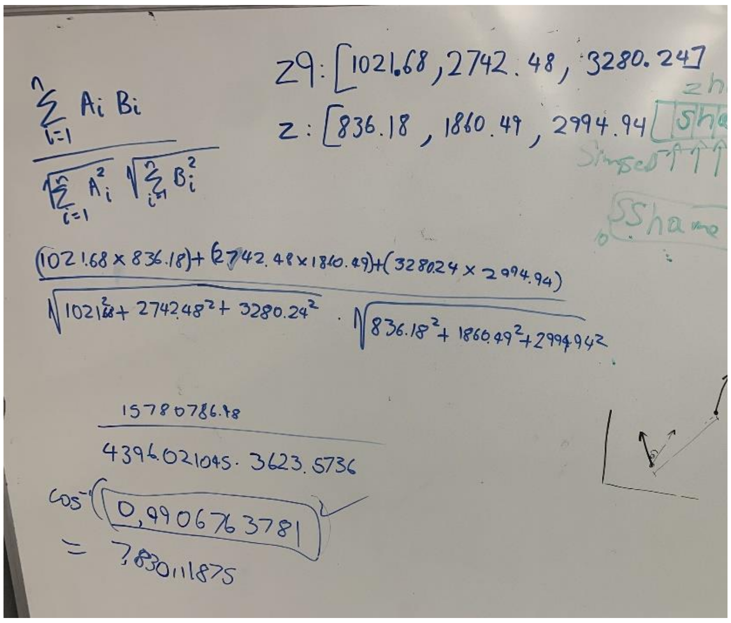
$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

$$\|\vec{a}\| = \sqrt{a_1^2 + a_2^2 + a_3^2 + \cdots + a_n^2}$$

$$\|\vec{b}\| = \sqrt{b_1^2 + b_2^2 + b_3^2 + \cdots + b_n^2}$$

Figure 9. Cosine similarity formula

We'll be comparing the distance between the input and every vector on the 1st depth to find the smallest one, like so:



Z9: [1021.68, 2742.48, 3280.24]
z: [836.18, 1860.49, 2994.94]
$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$
$\frac{(1021.68 \times 836.18)+(2742.48\times1860.49)+(3280.24\times2994.94)}{\sqrt{1021.68^2+2742.48^2+3280.24^2}\cdot\sqrt{836.18^2+1860.49^2+2994.94^2}}$
$\frac{15780786.48}{4394.021045\cdot3623.5736}$
$\cos^{-1}(0.9906763781)$
$= 7.830111875$

Figure 10. Cosine similarity mathematical application

We could choose to compare the similarity, distance, radian angle, or even degree angle, but we'll be using cosine distance in this case which is:

$$1 - cos^{-1}(\emptyset)$$

with ∅=

$$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Figure 11. cosine similarity formula

3.6. Vector space representation

We observe the representation of what this comparison looks like on a plane:

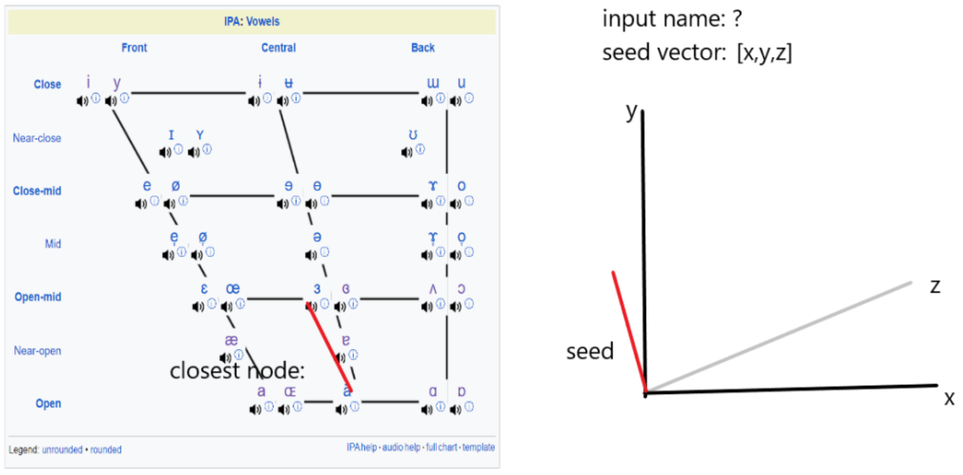


Figure 12. Superimposition of input vector on IPA Chart

3.7. Cosine similarity implementation

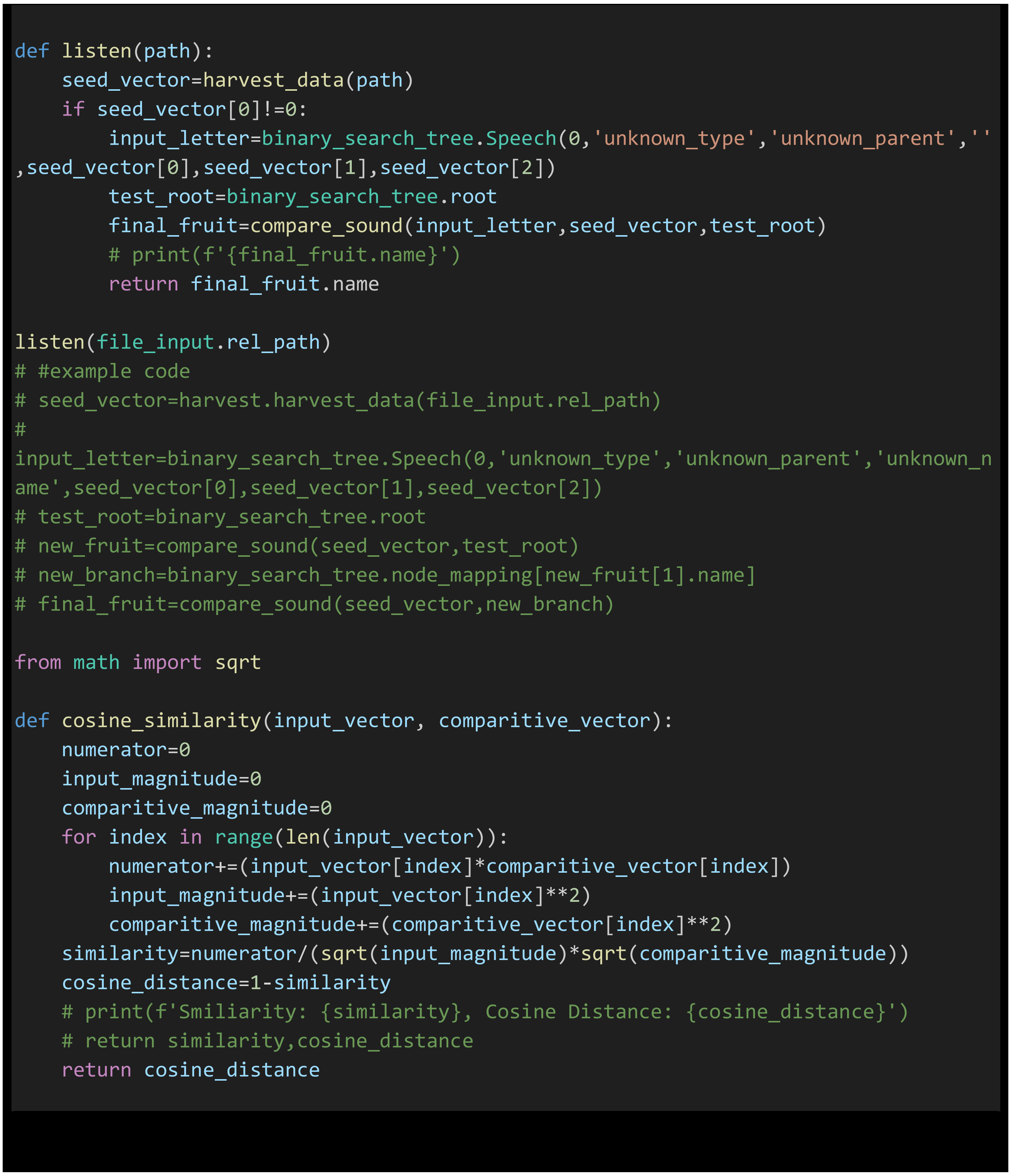
For each node in the first depth we will compare the cosine similarity to the input vector, the closets one will be the second depth to which we go and repeat the same operation. The commented-out code illustrates the trial and error it took to come to these conclusions as to which way is best. in an attempt to be more space efficient, I tried to heap sort as I went but the result of that was always the midpoint rather than the minimum, so I opted to use the default python method sorted. You can use a sorting algorithm of your choice like merge or selection sort which have a time complexity of O(nlogn) and $O(n^2)$ respectively and space complexity of O(mn) and O(1).

compare3.py

```
def compare_sound(input_letter,seed_vector,harvest_root):
    #1/2 depths
    stem=harvest_root.get_children_data()
    leaf=[]
    for branch in stem:
        petiole=[branch.f1,branch.f2,branch.f3]
        vein=cosine_similarity(seed_vector,petiole)
        branch.set_speech_cost(vein)
        # # # sort as we go
        # heapq.heappush(leaf, (branch.cost, branch))
        # #remove the other branch if it is bigger, to save memory
        # if len(leaf)>1:
        #     heapq.heappop(leaf)
        leaf.append((branch))
    leaf=sorted(leaf,key=lambda x:x.cost)
    fruit = leaf[0]
    # print(f'Most similar sound 1/2: {fruit[1].print_speech()} distance:
{fruit[0]}, vein: {vein}')
    # print(f'Most similar sound 1/2: {fruit.print_speech()} distance:
{fruit.cost}, vein: {vein}')

    #2/2 depths
    new_root=binary_search_tree.node_mapping[fruit.name]
    stem=new_root.get_children_data()
    leaf=[]
    for branch in stem:
        petiole=[branch.f1,branch.f2,branch.f3]
        vein=cosine_similarity(seed_vector,petiole)
        branch.set_speech_cost(vein)
        # # sort as we go
        # heapq.heappush(leaf, (vein, branch))
        # #remove the other branch if it is bigger, to save memory
        # if len(leaf)>1:
        #     heapq.heappop(leaf)
        leaf.append((branch))
    leaf=sorted(leaf,key=lambda x:x.cost)
    new_fruit=leaf[0]
    # Now, leaf_sorted contains tuples sorted based on the vein value
    # print(f'Most similar sound 2/2: {new_fruit[1].print_speech()}, distance:
{new_fruit[0]}, vein: {vein}')
    # print(f'Most similar sound 2/2: {new_fruit.print_speech()}, distance:
{new_fruit.cost}, vein: {vein}')
    return new_fruit

# def listen(path):
#     seed_vector=harvest_data(path)
#     input_letter=binary_search_tree.Speech(0,'unknown_type','unknown_parent','',s
eed_vector[0],seed_vector[1],seed_vector[2])
#     test_root=binary_search_tree.root
#     final_fruit=compare_sound(input_letter,seed_vector,test_root)
#     print(f'{final_fruit[1].name}')
```



```
def listen(path):
    seed_vector=harvest_data(path)
    if seed_vector[0]!=0:
        input_letter=binary_search_tree.Speech(0,'unknown_type','unknown_parent',''
,seed_vector[0],seed_vector[1],seed_vector[2])
        test_root=binary_search_tree.root
        final_fruit=compare_sound(input_letter,seed_vector,test_root)
        # print(f'{final_fruit.name}')
        return final_fruit.name

listen(file_input.rel_path)
# #example code
# seed_vector=harvest.harvest_data(file_input.rel_path)
#
input_letter=binary_search_tree.Speech(0,'unknown_type','unknown_parent','unknown_n
ame',seed_vector[0],seed_vector[1],seed_vector[2])
# test_root=binary_search_tree.root
# new_fruit=compare_sound(seed_vector,test_root)
# new_branch=binary_search_tree.node_mapping[new_fruit[1].name]
# final_fruit=compare_sound(seed_vector,new_branch)

from math import sqrt

def cosine_similarity(input_vector, comparitive_vector):
    numerator=0
    input_magnitude=0
    comparitive_magnitude=0
    for index in range(len(input_vector)):
        numerator+=(input_vector[index]*comparitive_vector[index])
        input_magnitude+=(input_vector[index]**2)
        comparitive_magnitude+=(comparitive_vector[index]**2)
    similarity=numerator/(sqrt(input_magnitude)*sqrt(comparitive_magnitude))
    cosine_distance=1-similarity
    # print(f'Smiliarity: {similarity}, Cosine Distance: {cosine_distance}')
    # return similarity,cosine_distance
    return cosine_distance
```

Figure 13. Comparison between nodes by cosine similarity

3.8. Slicing and Comparing

Having worked on and tested the viability of a single comparison it is possible to check multiple inputs' similarity to the tree nodes, it just involves slicing.

This is not a very hard concept, we simply cut the word or sentence into intervals of no less than 30ms. It is up to you to decide which interval provides the best results and tune the model accordingly.

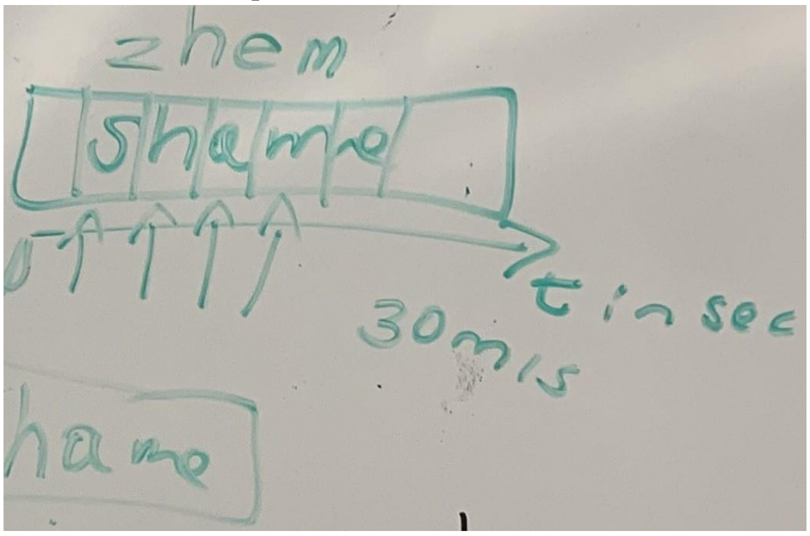


Figure 14. Slicing whiteboarding example

Once that's done you are ready to compare. Once the comparisons are done, the program outputs the letters it thinks were said and it could be taken a step further by implementing a spell-checking algorithm to find the nearest word match to the letters that were returned.

That's it! The program now attempts to transcribe audio files of any size in a simpler manner than traditional methods through phonetic partitioning. Slicing the input is partitioning and the comparisons are done phonetically at the first depth. In the next section we discuss the output.

4. Results

The results were indicative that the formant values in the dataset were not as accurate as they could be. That is primarily due to the lack of availability for single phoneme datasets in the English language. The formant values shown were harvested from Wikipedia's IPA consonant and vowel pages respectively.

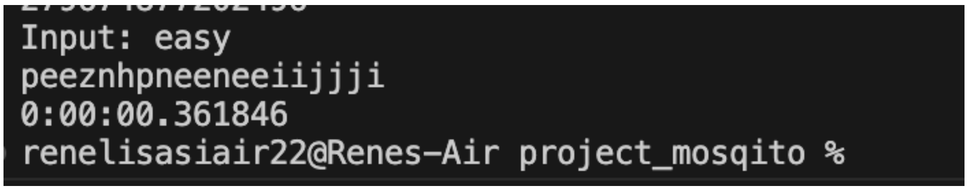
Another drawback is cosine similarity has one drawback in our case that is, the vowels are primarily determined by f1, consonants are primarily determined by f3, yet cosine similarity blends the two

The size of harvested data ranged around 10-30ms, with some not even having a peep during playback.

Finally, it is inconclusive whether or not the novel implementation and slicing strategy is any faster, accurate or would need less training than existing audio transcription libraries, however, it did prove to be more space efficient and lightweight than both CMU-Sphinx and Google's Speech Recognition.

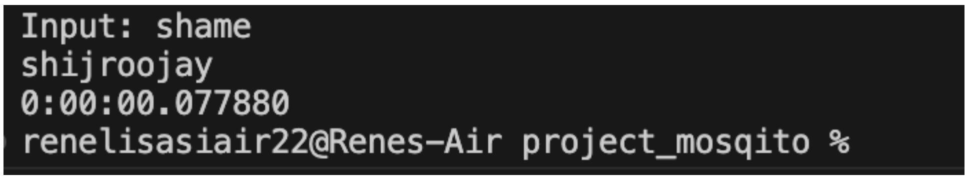
In the end this implementation is able to transcribe audio from file but would need further development to become fully fleshed. It serves as a very good basis for further research. Keep in mind it took CMU-Sphinx 20 years to get to the same level of accuracy from audio file as mine which was done in 4 months.

Output 1: ‘easy’



```
Input: easy
peeznhpneeneeiijjji
0:00:00.361846
renelisasiair22@Renes-Air project_mosqito %
```

Output 2: ‘shame’



```
Input: shame
shijroojay
0:00:00.077880
renelisasiair22@Renes-Air project_mosqito %
```

5. Conclusions

This study involved finding a new, simple way to transcribe audio. All other existing technologies are either specialized in direct speech or commercial software. This implementation focused on the use of cosine similarity to compare slices of audio file input for further use. Many research papers focus on music, animal sounds, and other use cases further and further away from the need to bridge the gap between organic human data and self-digitization. Thus, it is important to emphasize the need for studies to focus on the development of human data recognition for uses such as direct speech between humans and robots, eliminating the need for peripheral communication.

References

3. Macquarie University. "The Waveforms of Speech." Department of Linguistics, Macquarie University, www.mq.edu.au/about/about-the-university/our-faculties/medicine-and-health-sciences/departments-and-centres/department-of-linguistics/our-research/phonetics-and-phonology/speech/acoustics/speech-waveforms/the-waveforms-of-speech.

4. EduHK. "Formants of Vowels." English Pronunciation Learning Centre, EduHK, corpus.eduhk.hk/english_pronunciation/index.php/2-2-formants-of-vowels/.

5. University of Manitoba. "IPA Consonants." Department of Linguistics, University of Manitoba, home.cc.umanitoba.ca/~krussll/phonetics/ipa/ipa-consonants.html.

6. Syntheticspeech. "How to Extract Formant Tracks with Praat and Python." Syntheticspeech Blog, blog.syntheticspeech.de/2021/03/10/how-to-extract-formant-tracks-with-praat-and-python/.

7. Research India Publications. "Formant Frequency Estimation of Vowels Using Linear Predictive Coding." International Journal of Applied Engineering Research, vol. 13, no. 12, 2018, pp. 10217-10222, www.ripublication.com/ijaer18/ijaerv13n12_107.pdf.

8. VoxForge. "Create Acoustic Models with HTK." VoxForge, www.voxforge.org/home/dev/acousticmodels/linux/create/htkjulius/tutorial.

9. Loredana Cirstea. "DNN Speech Recognizer." GitHub Notebook, notebook.community/loredanacirstea/ai-algos/DNN%20Speech%20Recognizer/vui_notebook.

10. University of Edinburgh. "Introduction to Automatic Speech Recognition." School of Informatics, University of Edinburgh, www.inf.ed.ac.uk/teaching/courses/asr/2019-20/asr01-intro.pdf.

11. Mael Fabien. "Speech Recognition Using Deep Neural Networks." Mael Fabien's Blog, maelfabien.github.io/machinelearning/speech_reco/#.

12. Mael Fabien. "Gaussian Mixture Model." Mael Fabien's Blog, maelfabien.github.io/machinelearning/GMM/#.

13. ResearchGate. "How can I estimate a person's vocal tract length using a recorded audio file?" ResearchGate, www.researchgate.net/post/How-can-I-estimate-a-persons-vocal-tract-length-using-a-recorded-audio-file#:~:text=The%20first%20formant%20F1%20is,is%20the%20speed%20of%20sound.

14. YouTube. "Vocal Tract Length (VTL) Measurement." Speech and Hearing Science Channel, www.youtube.com/watch?v=PYlr8ayHb4g.