Computational Cognitive Neuroscience of Language Laboratory
Department of Psychological Sciences and CT Institute for the Brain and Cognitive Sciences,
University of Connecticut

Methods Report 2017.01 *Revision 0.2, 2017.09.20*

TISK 1.0 DISTRIBUTION

Heejo You, Thomas Hannagan, & Jim Magnuson
Contact: james.magnuson@uconn.edu

This document describes the Python distribution of TISK, the Time Invariant String Kernel model of spoken word recognition (Hannagan, Magnuson, & Grainger, 2013). The distribution is available from a github repository: https://github.com/CODEJIN/Tisk. This version is based closely on the original code developed by Thomas Hannagan, but was completely reimplemented in 2016-17 by Heejo You (with substantial programming advice from Thomas Hannagan and minor usability input from Jim Magnuson). This document walks through installation and basic use of the script. Note that the code is distributed as-is, with no promise of support or guarantee that the code is free of errors. Comments and bug reports are welcome.

TABLE OF CONTENTS

1. OVERVIEW OF TISK

In all likelihood, since you have found this file, you are already familiar with the TISK model. We will go through a very brief overview of the model. Skip to §2 if you don't need this overview.

In TISK, there are 4 sets of units (see Figure 1).

1. **Phoneme inputs**. These are time-specific nodes, with a copy of each phoneme input node for each time slot. By default, there are 10 time slots and 14 phonemes. Phoneme input nodes have weighted connections to diphone and single-phone nodes in the n-phone layer. Phoneme-input to diphone connections are somewhat complex; see Hannagan et al. (2013) for details.
2. **Diphone nodes**. These are time-invariant nodes (that is, just one instance of each) that correspond to all ordered phoneme pairs possible from the phoneme inventory. These receive input from appropriate phoneme input nodes and send outputs to appropriate words (e.g., /kæt/ [CAT] will get input from /kæ/, /kt/, and /æt/).
3. **Single-phone nodes**. These are also time-invariant nodes, with one instance for each of the phonemes in the phoneme inventory. These receive input from appropriate phoneme input nodes and send outputs to appropriate words (e.g., /kæt/ [CAT] will get input from /k/, /æ/, and /t/).
4. **Word nodes**. These are also time-invariant nodes, with one instance for every word in the lexical inventory. These receive input from appropriate diphone and single-phone nodes at the n-phone level. The word level also includes lateral inhibition; each word node can activate any other word node.

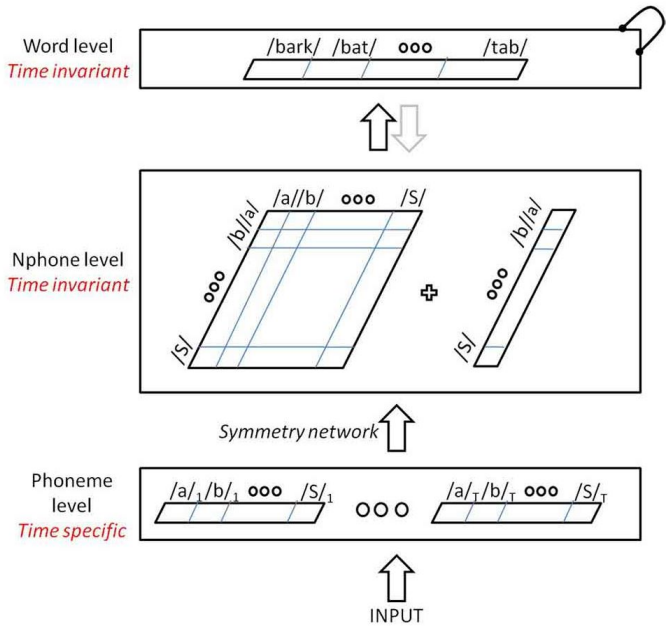

Figure 1: TISK structure (from Figure 3 of Hannagan et al., 2013).

2. DOWNLOADING AND INSTALLING

TISK 1.0 can be downloaded from its github repository: https://github.com/CODEJIN/Tisk. TISK is implemented as a single Python 3 script. You must acquire Python fundamentals on your own, and install Python 3. TISK imports some standard libraries (time, os) but also requires 2 additional libraries that you must install:

1. Numpy
2. Matplotlib.pyplot

Note that if you just install the free version of Anaconda with Spyder, it comes pre-installed with these libraries. We have confirmed that a Python novice (with some familiarity with 'IDEs' [integrated development environments] from RStudio) was able to get TISK up and running under Spyder just from this document.

3. FILES

TISK 1.0 requires 3 files:

1. **Basic_TISK_Class.py** This is the core Python code for TISK.
2. **Pronunciation_Data.txt** This file corresponds to the original 212-word TRACE lexicon (McClelland & Elman, 1986) that was used in the original TISK paper (Hannagan et al., 2013). You can simply edit or replace this file to change the lexicon.
3. **Phoneme_Data.txt** Optional file. The phoneme list is initially generated as all unique phonemes included in the words in '**Pronunciation_Data.txt**'. However, in unusual cases, users may wish to include additional phonemes not included in any word. To allow for this possibility, you can expand the set of phonemes by adding a phoneme to this file (Phoneme_Data.txt). Note that Basic_TISK_Class.py expects phonemes to be represented by single characters, and is is case sensitive. It is best to stick to ASCII characters. So, for example, you wanted to add `/æ/`, you could use `/A/`.

4. RUNNING TISK

4a. Set up your environment

4a.1. Anaconda (platform independent).

Download Anaconda. Follow the installer instructions for your platform. As of this date (March, 2017), Spyder is a default option within Anaconda. This is an IDE for Python that is already somewhat customized for scientific purposes. It comes with the `numpy` and `matplotlib.pyplot` libraries mentioned above. Once you have it installed, launch Spyder. Use the menus or icons to open a file. Navigate to the directory where you have placed the TISK files. Set the working directory to that location. Prepare to enter commands in 4b into the Console window.

4a.2. Windows.
1. Open the console. (cmd in Run menu of Start)
2. Type `'python'` or `'ipython'` (recommended: `'ipython'`)

*4a.3. Macintosh/Linux/*nix terminal or Python interpreter.*
1. Open the terminal.
2. Type `'python'` or `'ipython'` (recommended: `'ipython'`)

4b. Import TISK and initialize the model
Type the following commands (lines preceded by "#" are comments and can be skipped, but can also be pasted into the python interpreter, as they will be ignored):

```
# load the TISK functions
import Basic_TISK_Class as tisk

# load the phoneme and pronunciation [word] lists and
# prepare appropriate connections
phoneme_List, pronunciation_List = tisk.List_Generate()

# initialize the model with the the phoneme_List,
# pronunciation_List, number of time slots, and threshold
tisk_Model = tisk.TISK_Model(phoneme_List, pronunciation_List,
                      time_Slots = 10, nPhone_Threshold = 0.91)
```

Ten times slots are enough for the words in the default "slex" pronunciation_List. However, if you have a lexicon with longer words (more than 10 phonemes), you can increase the value of `time_Slots` or just leave this parameter out altogether (just put a ")" after `pronuncation_List` and end the command there), and `time_Slots` will automatically be set to the length of the longest word in `pronunciation_List`. However, there are cases where you may wish to present phoneme series longer than the longest word (e.g., to present long nonwords or to present series of words). In those cases, you need to set `time_Slots` appropriately.

4c. Initialize or modify parameters
Before the simulation, you must initialize the parameters. To use the default parameters of TISK 1.0 (Hannagan et al., 2013), just enter:

```
tisk_Model.Weight_Initialize()
```

The model does not automatically initialize because this is the step where all connections are made, etc., and initialization can take a long time if you have a large model with thousands of

words. To control specific categories of parameters or specific parameters, you can use the following examples:

```
tisk_Model.Decay_Parameter_Assign(
                    decay_Phoneme = 0.001,
                    decay_Diphone = 0.001,
                    decay_SPhone = 0.001,
                    decay_Word = 0.01)
tisk_Model.Weight_Parameter_Assign(
                    input_to_Phoneme_Weight = 1.0,
                    phoneme_to_Phone_Weight = 0.1,
                    diphone_to_Word_Weight = 0.05,
                    sPhone_to_Word_Weight = 0.01,
                    word_to_Word_Weight = -0.005)
tisk_Model.Feedback_Parameter_Assign(
                    word_to_Diphone_Activation = 0,
                    word_to_SPhone_Activation = 0,
                    word_to_Diphone_Inhibition = 0,
                    word_to_SPhone_Inhibition = 0)
tisk_Model.Weight_Initialize()
```

If you only want to modify a subset of the parameters in these sets, just specify the ones you want to change, and the others will retain their current values. For example:

```
tisk_Model.Decay_Parameter_Assign(
                    decay_Phoneme = 0.002,
                    decay_Diphone = 0.0005)
tisk_Model.Weight_Initialize()
```

4d. Parameter details

If you need to see the current parameters, you can type the following command:

```
tisk_Model.Parameter_Display()
```

Note that if you use this command before initializing model, you will just see a message asking you to initialize. When parameters are initialized, this command will return a list like this:

```
nPhone_Threshold: 0.91
iStep: 10
time_Slots: 10
Phoneme_Decay: 0.001
Diphone_Decay: 0.001
SPhone_Decay: 0.001
Word_Decay: 0.01
Input_to_Phoneme_Weight: 1.0
Phoneme_to_Phone_Weight: 0.1
Diphone_to_Word_Weight: 0.05
SPhone_to_Word_Weight: 0.01
Word_to_Word_Weight: -0.005
Word_to_Diphone_Activation_Feedback: 0.0
Word_to_SPhone_Activation_Feedback: 0.0
Word_to_Diphone_Inhibition_Feedback: 0.0
Word_to_SPhone_Inhibition_Feedback: 0.0
```

The table on the following page explains each of these parameters.

Table 1: TISK parameters.

Parameter	Default	Description
nPhone_Threshold	0.91	N-Phone nodes only send activation to words when their activation exceeds this threshold.[1]
iStep	10	The number of activation cycles between time slots (e.g., if iStep is set to 10, the first phoneme will be presented [with decay, etc.] from cycles 0-9, the second from 10-19, etc.).
time_Slots	10	Decide the maximum length of words. Time slots will be spaced iStep cycles apart.[2]
Phoneme_Decay	0.001	Governs how much activation from previous time step is retained for phoneme input nodes.
Diphone_Decay	0.001	Decay for diphone nodes.
SPhone_Decay	0.001	Decay for single phone nodes in the diphone layer.
Word_Decay	0.01	Decay for word nodes.
Input_to_Phoneme_Weight	1	Gain from input pattern to phoneme input nodes.
Phoneme_to_Phone_Weight	0.1	Gain: phoneme inputs to appropriate single-phonemes.
Diphone_to_Word_Weight	0.05	Gain from diphones to words containing them.
SPhone_to_Word_Weight	0.01	Gain from single-phonemes to words containing them.
Word_to_Word_Weight	-0.005	Lateral inhibition.
Word_to_Diphone_Activation_Feedback	0	Positive weights from words to constituent diphones.[3]
Word_to_SPhone_Activation_Feedback	0	Positive weights from words to constituent single phonemes.
Word_to_Diphone_Inhibition_Feedback	0	Weights from words to *non*-constituent[4] diphones.
Word_to_SPhone_Inhibition_Feedback	0	Weights from words to *non*-constituent single phonemes.

[1] This threshold relates importantly to time_Slots, and changing time_Slots will lead to a warning and recommendation to adjust parameters to conform to the following formula [iStep x (time_Slots – 1) + 1] / [iStep x time_Slots] when (Phoneme_to_Phone_Weight x time_Slots) ≤ nPhone_Threshold.

[2] Thus, a simulation will have IStep_Length x Time_Slot_Length cycles. So if IStep_Length is 10 and Time_Slot_Length is 8, phone 1 would be sustained over cycles 0-9, phone 2 over cycles 10-19, and phone 8 over cycles 70-79.

[3] Note that feedback was not used in Hannagan et al. (2013), and adding feedback will likely require modifying other parameters to make the model stable.

[4] The "inhibition feedback" nodes are meant for top-down inhibition (consistent, e.g., with the Cohort model). However, these are just connections to diphones or phones *not* contained in a word. If you set this parameter to a positive value, you would actually make a word activate all sublexical units it does not contain!

Changing parameters. §4c shows how to change several of these variables. Others must be modified by changing the **Basic_TISK_Class.py** file directly.

5. BASIC METHODS FOR TISK SIMULATIONS, INCLUDING GRAPHS & DATA EXPORT

5a. Simulate processing of a phoneme string and graph results for phonemes and words

Here is an example, basic command that calls a simulation of the word 'pat' (technically, it is more correct to say "a simulation of the pronunciation 'pat'", as you can specify pronunciations that are not in the lexicon [i.e., the `pronunciation_List`]):

```
tisk_Model.Display_Graph(pronunciation='pat')
```

Specifying a pronunciation with this command triggers a simulation with that phoneme sequence as the input. You can specify any arbitrary sequence of phonemes (that is, the sequence does not have to be a word), as long as the phonemes are in the `phoneme_List`, and the sequence length does not exceed the maximum length.

On its own, this command doesn't do anything apparent to the user (though the simulation is in fact conducted). To create a graph that is displayed within an IDE, you can add arguments to display specific phonemes:

```
tisk_Model.Display_Graph(
          pronunciation='pat',
          display_Phoneme_List = [('p', 0), ('a', 1), ('t',
2)])
```

This code means "input the pronunciation /pat/, and export a graph with phoneme activations for /p/, /a/, and /t/ in the first, second, and third positions, respectively". 'Phoneme' is the label that specifies the phonemic **input** units (see §1). Since the phoneme input nodes are duplicated at each time slot, you need to specify a time slot when you specify a phoneme input node. The command above creates a graph like this (displayed in some IDEs, like Spyder):

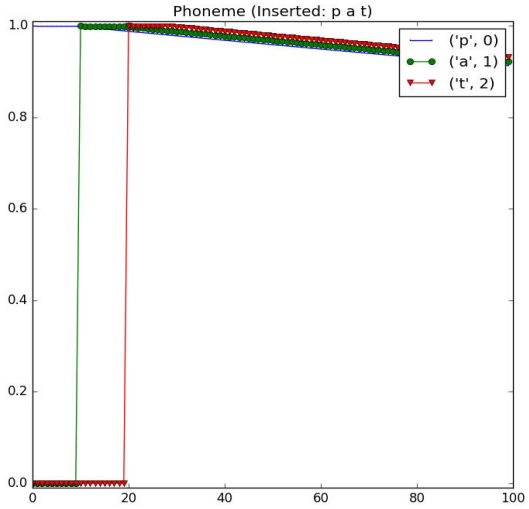

Figure 2: Phoneme activation plot example

We can extend this to create activation graphs of diphones, single phones, and words. The following example creates one of each:

```
tisk_Model.Display_Graph(pronunciation='pat',
                         display_Diphone_List = ['pa', 'pt', 'ap'],
                         display_Single_Phone_List = ['p', 'a', 't'],
                         display_Word_List = ['pat', 'tap'])
```

When you run this code, you will see the three graphs below. You can specify arbitrarily many items in any such command.

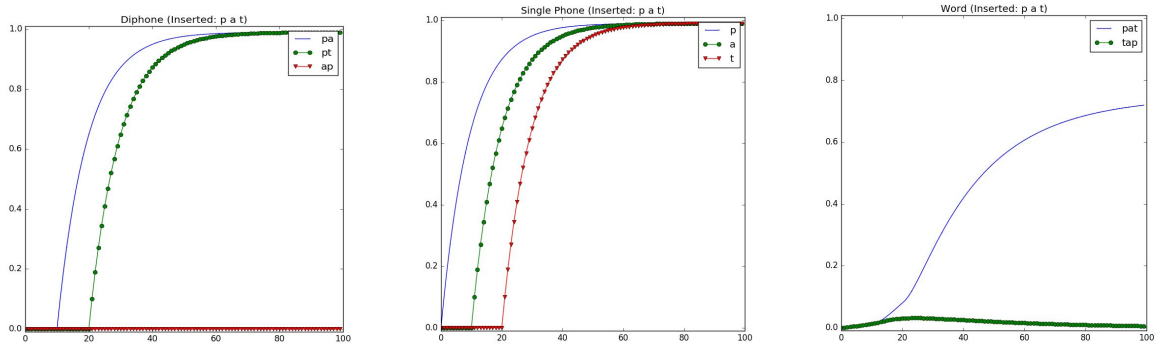

Figure 3: N-Phone and word activation plot example

The graphs that are produced are fairly basic and may not suffice for publication. You could tweak the code to adjust the graphs to your liking, but a more typical procedure would be to export the underlying data and create publication-ready graphs using other software, such as R. Let's next look at the methods for extracting data.

5b. Export graph results as PNG files

To export the result as a graphic file (PNG format), we add a parameter:

```
tisk_Model.Display_Graph(pronunciation='pat',
                    display_Diphone_List = ['pa', 'pt', 'ap'],
                    display_Single_Phone_List = ['p', 'a', 't'],
                    display_Word_List = ['pat', 'tap'],
                    file_Save = True)
```

When you run this code, the three graphs will be saved as "p_a_t.Diphone.png", "p_a_t.Single_Phone.png", and "p_a_t.Word.png".

5c. Extract simulation data a numpy matrix

The basic method for extracting data as a numpy matrix is as follows:

```
tisk_Model.Extract_Data(pronunciation='pat')
```

The method is similar to the graphing functions. The code above is the core command that readies appropriate structures for extraction, but it does not by itself generate any result for the user. To extract data, you need to provide arguments specifying what you want. To get data corresponding to the word plot above (showing the activations of /pat/ and /tap/ given the input /pat/), you would use the following command to put the data in a numpy matrix called 'result':

```
result = tisk_Model.Extract_Data(pronunciation='pat',
                            extract_Word_List = ['pat', 'tap'])
```

When you run this, the 'result' variable becomes a list with length 1, consisting of a single numpy matrix with shape (2, 100). The first and second rows are the activation patterns of the word units for /pat/ and /tap/, respectively.

Let's try a more complex example:

```
result = tisk_Model.Extract_Data(pronunciation='pat',
       extract_Phoneme_List = [('p', 0), ('a', 1), ('t', 2)],
       extract_Single_Phone_List = ['p', 'a', 't'])
```

Here, 'result' becomes a list with length 2. The first item is a numpy matrix with the input unit activations for the 3 specified phonemes across the 100 steps of the simulation. The second is a numpy matrix with the activations of the specified single phonemes in the n-phone layer over the 100 steps of the simulation.

5d. Export simulation data to text files

To export results to text files, we add a parameter:

```
result = tisk_Model.Extract_Data(pronunciation='pat',
    extract_Word_List = ['pat', 'tap'], file_Save=True)
```

This creates a text file called "p_a_t.Word.txt". The file has 102 columns and 2 rows. The first column is the input string ("pat"), and the second is the specified word to track (row 1 is "pat" and row 2 is "tap"). Columns 3-102 are activations for the corresponding word in cycles 0-99.

5e. Batch simulation of multiple words

Of course, we often want to conduct simulations of many words. First, here's an easy way to assess average performance on a specified list of items:

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = ['baks', 'bar', 'bark', 'bat^l',
'bi'])
```

When you run this code, the model will simulate the 5 words and check the reaction time (RT) and accuracy for each. The variable 'rt_and_ACC' will be a list of 6 items, with mean RT and accuracy for the specified words computed using three different methods (abs = based on an absolute threshold [target must exceed threshold], rel = relative threshold [target must exceed next most active item by threshold], tim = time-based threshold [target must exceed next most active item by threshold for at least timSteps cycles]):

`rt_and_ACC[0]`: Mean of RT_{abs}
`rt_and_ACC[1]`: Mean of ACC_{abs}
`rt_and_ACC[2]`: Mean of RT_{rel}
`rt_and_ACC[3]`: Mean of ACC_{rel}
`rt_and_ACC[4]`: Mean of RT_{tim}
`rt_and_ACC[5]`: Mean of ACC_{tim}

What if you want to evaluate mean accuracy and RT for every word in the current lexicon with the current parameter settings? To do this, you would modify the command as follows, where we specify the `pronunciation_List` to be the full `pronunciation_List`:

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = pronunciation_List)
```

You can also modify the parameters used for the different accuracy methods. The default criteria are: abs = 0.75, rel = 0.05, tim = 10 (timSteps). If you want to change the criteria you can do so like this:

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = ['baks', 'bar', 'bark', 'bat^l',
'bi'],
     absolute_Acc_Criteria=0.6,
     relative_Acc_Criteria=0.01,
     time_Acc_Criteria=5)
```

When you use this code, the criteria of abs, rel, and tim will be changed to 0.6, 0.01, and 5, respectively.

Often, you may want to obtain the RT values for each word in a list, rather than the mean values. You can do this using the `reaction_Time` flag with the `Run_List` procedure. Currently, this requires you to specify a file to write the data to (which could be read back in using standard Python techniques):

```
tisk_Model.Run_List(
     pronunciation_List = ['baks','bar','bark','bat^l','bi'],
     output_File_Name = "Test",
     reaction_Time=True)
```

This will create an output file named "Test_Reaction_Time.txt". Its contents would be:

```
Target     Absolute   Relative   Time_Dependent
baks       58         40         46
bar        84         28         33
bark       74         52         56
bat^l      60         39         46
bi         nan        23         13
```

Accuracy is indicated by the value for each word for each accuracy criterion. Items that were correctly recognized according to the criterion will have integer values (cycle at which the criterion was met). Items that were not will have values of "nan" (not a number, a standard designation for a missing value). In the current example, we can see that /bi/ ("bee") did not meet the absolute criterion.

If you wanted to find obtain the RTs for every word in your lexicon, you would replace the word list with pronunciation_List:

```
tisk_Model.Run_List(
     pronunciation_List = pronunciation_List,
     output_File_Name = "all",
     reaction_Time=True)
```

5f. Extract data for multiple words in text files

To export results with multiple words, you can use the 'Run_List' function again, as follows:

```
     rt_and_ACC = tisk_Model.Run_List(
          pronunciation_List = ['baks', 'bar', 'bark', 'bat^l',
'bi'],
          output_File_Name = 'Result',
          raw_Data = True,
          categorize=True)
```

When you run this code, you get text files with what we call 'raw' and 'category' outputs. In **raw files** (e.g., for this example, Result_Word_Activation_Data.txt), you get outputs that show you the activations for every word in the lexicon at every time step for each target specified. The file format is very simple. There is a 1-line header with column labels. The first column is 'target', the second is 'word', and the following columns are cycles 0-*C*, where *C* is the final cycle (which will have the value [(`time_Slots x iStep`) - 1]. So in a row that begins:

baks ad 0.0 0.0 ...

The target is 'baks' and this row will show you the activation of 'ad' when 'baks' (BOX) was the target over all time steps. To find the actual target activations, find the row that has 'baks' in the first two columns:

baks baks 0.0 0.0 0.001 0.002557431 0.0044149027022 ...

The phoneme file (e.g., Result_Phoneme_Activation.txt) has a similar structure, but adds the needed phoneme position column. Here are some examples for /b/ in positions 0-3 when the input is 'baks'. This shows how the phoneme activation ramps up slightly after insertion but then begins to decay:

Target	Phoneme	Position	0	1	2	3	4	5
baks	b	0	1	0.999	0.999001	0.999000999	0.999000999	0.999000999
baks	b	1	0	0	0	0	0	0
baks	b	2	0	0	0	0	0	0
baks	b	3	0	0	0	0	0	0

The diphone file (e.g., Result_Diphone_Activation.txt) simply gives you on each row the Target (word), a diphone, and then the activation of that diphone over time steps given that target as

input. The single phone file (e.g., Result_Single_Phone_Activation.txt) does the same thing for single phonemes from the n-phone layer.

In **category files** (e.g., for this example, Result_Category_Activation_Data.txt), for each target, you get the target activations for each word (e.g., for 'baks'):

Target	Category	0	1	2	3	4	5	6
baks	Target	0	0	0.001	0.002557	0.004415	0.006421	0.008485
baks	Cohort	0	0	0.001	0.002527	0.004313	0.006201	0.008103
baks	Rhyme	0	0	0	0	0	0	0
baks	Embedding	0	0	0	0	0	0	0
baks	Other	0	0	0.000141	0.000352	0.000591	0.000837	0.00108

In addition, for every word, this file includes the *mean* activations for different categories of items. These are:

Cohort: Words matching the target in the first two phonemes
Embedding: Words embedded in the target (e.g., AT is embedded in CAT)
Rhyme: Not really rhymes! Words mismatching the target only in first position.
Other: The mean of all other words (excluding Target, Cohorts, Embeddings, Rhymes).

The 'Other' category gives you a rough baseline for words unrelated to the target, even though it may still contain many related words (e.g., neighbors not included in cohorts, embeddings, and rhymes). As long as the lexicon is large, this provides a good baseline that should hover near 0.

You can also make graphs that correspond to the category data. For example, let's plot the category data for /baks/.

```
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List=['pat', 'tap', 'art'])
```

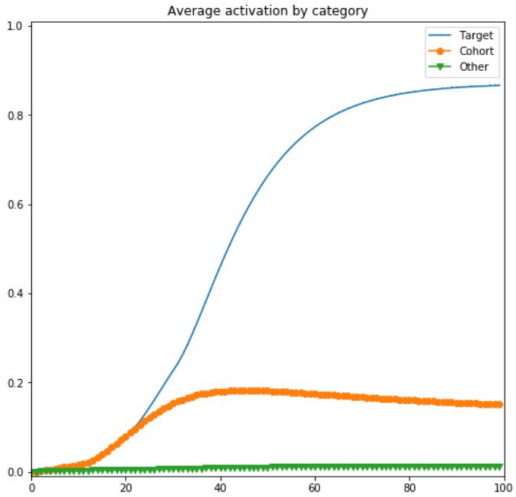

Figure 4: Average activation by category graph example 1

We can also get average data and average plots for a set of specified words. For example, suppose for some reason we were interested in the average category plot for the words /pat/, /tap/, and /art/ ("pot", "top", and "art"). We could call this command.

```
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List=['pat', 'tap', 'art'])
```

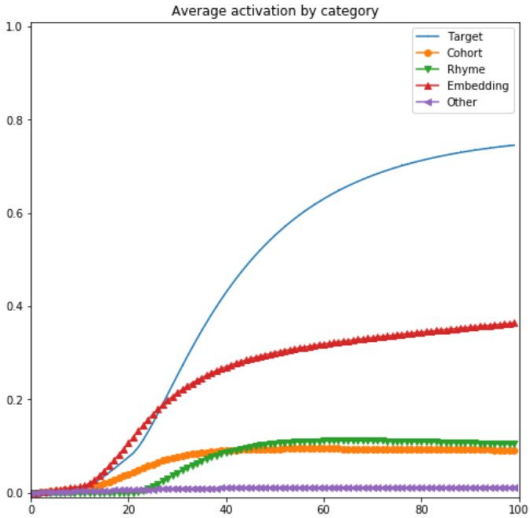

Figure 5: Average activation by category graph example 2

To save this graph as a PNG file, add the `file_Save=True` argument:

```
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List=['pat','tap', 'art'],
     file_Save=True)
```

By default, the graph associated with this command will be saved as "Average_Activation_by_Category_Graph.png". To specify a different filename (important if you wish, for example, to loop through many example sets in a Python script), you can do so as follows:

```
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List=['pat','tap', 'art'],
     output_File_Name='Result',
     file_Save=True)
```

In this case, the exported graph file name will become 'Result_Average_Activation_by_Category_Graph.png'. By setting the `output_File_Name` parameter, you can control the prefix of exported file name.

5g. Getting comprehensive data for every word in the lexicon

If we combine two recent examples, we can save activations for simulations of every word by replacing our pronunication_List argument above with pronunciation_List:

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = pronunciation_List,
     output_File_Name = 'all_words',
     raw_Data = True,
     categorize=True)
```

By extension, we can also generate a mean category plot over every word in the lexicon:

```
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List = pronunciation_List)
```

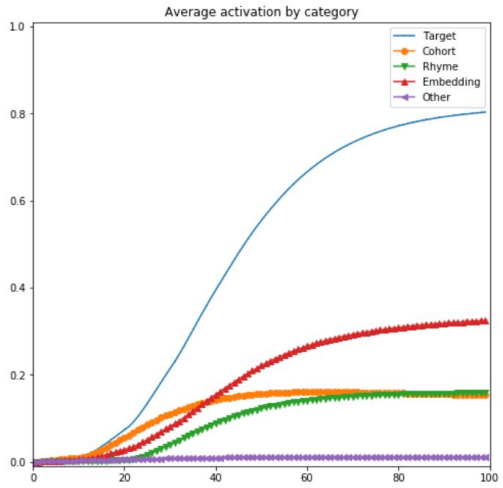

Figure 6: Average activation by category graph about all Slex pronunciations

---

5h. Batch size control

Depending on the size of your lexicon and the memory available on your computer, you may see the 'Memory Error' message when you run batch mode. Batch-mode simulation is not possible if the memory of the machine is too small to handle the size of the batch. To resolve this, you can use the `batch_Size` parameter to reduce the size of the batch. This parameter determines how many word simulations are conducted in parallel. It only controls the batch size, and does not affect any result. You will get the same result with any batch size your computer's memory can handle. The default value is 100. To see whether your computer memory can handle it, you can test larger values.

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = pronunciation_List,
     batch_Size = 10)

tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List=['pat','tap', 'art'],
     output_File_Name='Result', file_Save=True)
```

5i. Reaction time and accuracy for specific words

To check specific kinds of RT for specific words, you can use commands like these:

```
result = tisk_Model.Run('pat')
abs_RT = tisk_Model.RT_Absolute_Threshold(
                 pronunciation = 'pat',
                 word_Activation_Array = result[3],
                 criterion = 0.75)
rel_RT = tisk_Model.RT_Relative_Threshold(
                 pronunciation = 'pat',
                 word_Activation_Array = result[3],
                 criterion = 0.05)
tim_RT = tisk_Model.RT_Time_Dependent(
                 pronunciation = 'pat',
                 word_Activation_Array = result[3],
                 criterion = 10)
```

If TISK successfully recognized the inserted word, the reaction time will be returned. If the model failed to recognize the word, the returned value is 'numpy.nan'. Of course, you can change the criterion by modifying the parameter 'criterion'.

Of course, you could also get all accuracy and RT values for a specific word by using the command we introduced in §5e:

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = ['pat'])
```

5j. More complex simulations

Since TISK is implemented as a Python class, the user can do arbitrarily complex simulations by writing Python scripts. Doing this may require the user to acquire expertise in Python that is beyond the scope of this short introductory guide. However, to illustrate how one might do this, we include one full, realistic example here. In this example, we will compare competitor effects as a function of word length, by comparing competitor effects for words that are 3 phonemes long vs. words that are 5 phonemes long. All explanations are embedded as comments (preceded by "#") in the code below.

```
# first, select all words that have length 3 in the lexicon
length3_Pronunciation_List = [x for x in pronunciation_List if
     len(x) == 3]

# now do the same for words with length 5
length5_Pronunciation_List = [x for x in pronunciation_List if
     len(x) == 5]

# make a graph of average competitor effects for 3-phoneme
words
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List = length3_Pronunciation_List)

# make a graph and also save to a PNG file
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List= length3_Pronunciation_List,
     file_Save=True,
     output_File_Name='length_3_category_results.png')

# make a graph and also save to a PNG file
tisk_Model.Average_Activation_by_Category_Graph(
     pronunciation_List= length5_Pronunciation_List,
     file_Save=True,
     output_File_Name= 'length_5_category_results.png')

# save the length 3 data
tisk_Model.Run_List(
     pronunciation_List = length3_Pronunciation_List,
```

```
output_File_Name='length3data',
raw_Data = True, categorize = True)
```

Note that when you save data to text files, if you leave out the "`categorize = True`" argument, the file with word results will include the over time results for every target in pronunciation list. The first column will list the target, the second column the time step, and then there will be 1 column for every word in the lexicon (i.e., with the activation of that word given the current target at the specified time step).

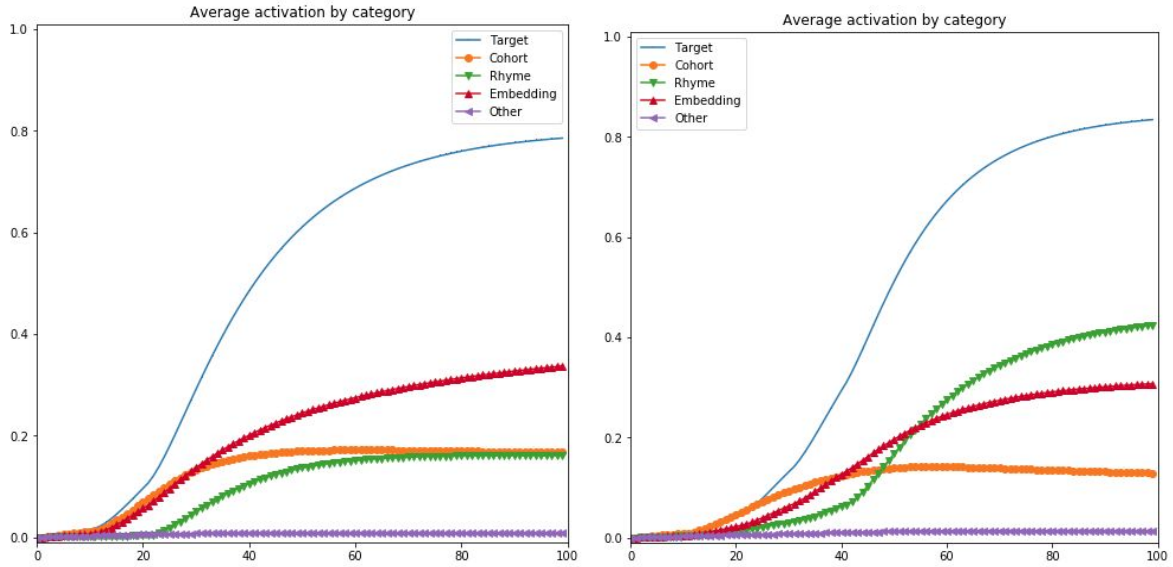

Figure 7: Average activation by category graph about filtered pronunciations

We see several interesting differences in Figure 7. First, 3-phoneme targets activate faster (e.g.,hitting a value of 0.4 about 10 cycles sooner than 5-phoneme words). On average, cohort effects appear to be weaker for longer words, and rhyme effects appear to be stronger. These results represent testable hypotheses about human SWR. They may also represent testable differences between models (e.g., if TRACE or some other model predicts little or no effect of word length on the magnitude of competitor effects).

Of course, this example just scratches the surface of what is possible since TISK is embedded within a complete scripting language. Using standard Python syntax, we can easily filter words by specifying arbitrarily complex conditions. Here are some examples:

```
# Select words with length greater than or equal to 3 with
# phoneme /a/ in position 2
filtered_Pronunciation_List = [x for x in pronunciation_List if
     len(x) >= 3 and x[2] == 'a']
# make a graph
tisk_Model.Average_Activation_by_Category_Graph(
```

```
        pronunciation_List = filtered_Pronunciation_List)

    # select words with length greater than or equal to 3 with
    # phoneme /a/ in position 2 and phoneme /k/ in position 3
    filtered_Pronunciation_List = [x for x in pronunciation_List if
        len(x) >= 3 and x[2] == 'a' and x[3]=='k']
```

6. ADVANCED SIMULATIONS AND ANALYSES

Since TISK is implemented as a Python class, you can do arbitrarily complex simulations by writing Python scripts. Doing this will require you to acquire expertise in Python that is beyond the scope of this short introductory guide. Good luck!