PyParse: A semiautomated system for scoring spoken recall data

ALEC SOLWAY
Princeton University, Princeton, New Jersey

AARON S. GELLER
Technion–Israel Institute of Technology, Technion City, Israel

PER B. SEDERBERG
Princeton University, Princeton, New Jersey

AND

MICHAEL J. KAHANA
University of Pennsylvania, Philadelphia, Pennsylvania

Studies of human memory often generate data on the sequence and timing of recalled items, but scoring such data using conventional methods is difficult or impossible. We describe a Python-based semiautomated system that greatly simplifies this task. This software, called PyParse, can easily be used in conjunction with many common experiment authoring systems. Scored data is output in a simple ASCII format and can be accessed with the programming language of choice, allowing for the identification of features such as correct responses, prior-list intrusions, extra-list intrusions, and repetitions.

Much of our knowledge concerning human memory comes from asking participants to recall a list of studied items either freely or in some prescribed order (e.g., forward or backward). In most of these studies, researchers ask participants to write their responses on paper for subsequent scoring and analysis. Using modern computers, which have come to dominate psychological experimentation over the last 25 years, one can also easily record participants' typed responses and use the computer to assist in scoring the recall protocols for both accuracy and order. Although collecting responses using a computer keyboard has advantages over using hand-written responses, spoken recall provides even further benefits.

With spoken recall, participants can respond more naturally and quickly, leading to a purer assay of memory function. Having participants type their responses forces them to actively engage with both the keyboard and the computer screen as they attempt to minimize their spelling mistakes. This interaction can result in marked interference. Furthermore, spoken recall lends itself more easily to measuring interresponse latency data, which has proven valuable in testing various theories of memory function (Kahana, 1996; Murdock & Okada, 1970; Patterson, Meltzer, & Mandler, 1971; Pollio, Richards, & Lucas, 1969; Polyn, Norman, & Kahana, 2009; Rohrer & Wixted, 1994; Wingfield, Lindfield, & Kahana, 1998). With keyboard responses, retrieval is often assumed to coincide with the first keystroke, but this is not always true. For instance, a participant may remember that a word on the list begins with the letter "S" (and press "S" on the keyboard), but not recall the rest of the word for some time thereafter. This adds noise to the latency data and makes subsequent analysis more difficult.

In view of these advantages, it may seem surprising that most modern researchers still rely on written or typed responses rather than on spoken recall. We believe that this is largely a consequence of the technical difficulties of scoring spoken recall using existing software. For example, consider the free recall task. After studying a list of items (typically words), participants are asked to recall the items in any order. Figure 1 illustrates a digitized recording of a sequence of spoken words. The approximate onset of each word is shown along with its identity. When presented with a large data set, locating the onset of each word with a high degree of accuracy and consistency is a formidable task. There is also no standard way of storing this information so that it is later easily accessible for analysis.

When faced with the challenge of scoring interresponse times in a spoken recall study in 1992, one of the authors of the present article (M.J.K.) began developing a set of software libraries to help run experiments involving spoken responses and to score the resulting data. After several generations of programming languages and numerous collaborators, this effort resulted in the development of the Python Experimental Programming Library (PyEPL), described in

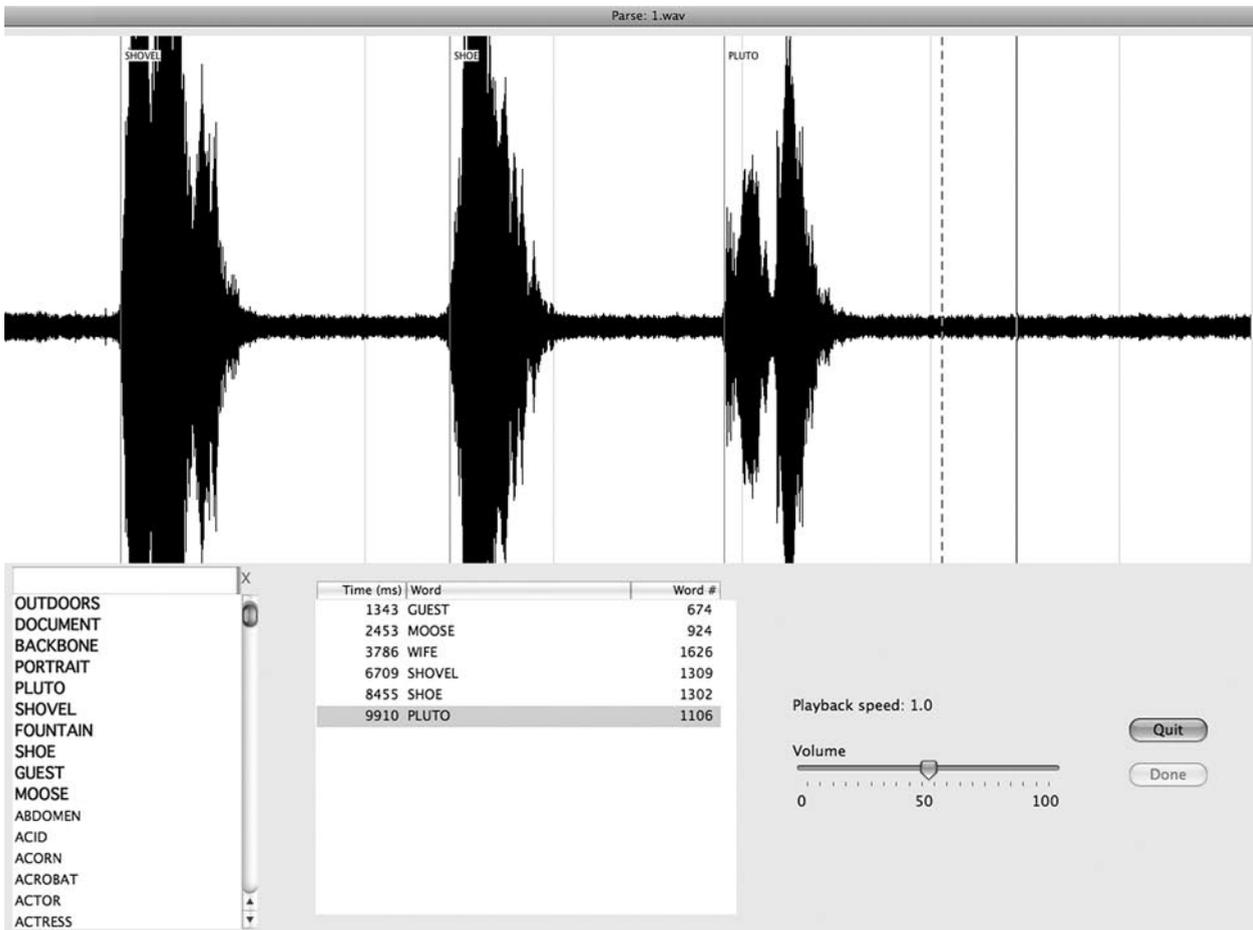**M. J. Kahana, kahana@psych.upenn.edu**

Figure 1. A typical PyParse session. The top half of the screen displays the waveform of a previously recorded study. The bottom half contains, from left to right, the response box and word pool, a list of the vocalizations marked so far along with their corresponding onsets, a volume slider, a playback speed display, and command buttons for closing the application in one of two ways (both of which are explained in the text).

Geller, Schleifer, Sederberg, Jacobs, and Kahana (2007), and the Python-based Recall Parser (PyParse), described in this article. We illustrate how the PyParse software can be used to rapidly score recall data for the sequence of responses and interresponse times while simultaneously maintaining a high level of accuracy and consistency. PyParse takes as input the audio files recorded during the course of an experiment, along with the list of presented words. It can thus be used with any experiment-authoring software that provides function calls for recording and storing digitized speech, including E-Prime, Psychtoolbox, PsyScript, and PyEPL.

Note that, in general, PyParse does not perform speech recognition on the recorded files. Although one could conceivably develop a speaker-independent voice recognition system for use in large verbal recall studies, our preliminary investigation of such technology suggests that much work is needed before such systems achieve the stringent accuracy requirements of memory studies. However, PyParse can automatically label the data from experiments involving a relatively small set of valid responses (e.g., recognition and confidence judgment experiments) with greater than 99% accuracy.

USAGE

The PyParse software, along with installation instructions and documentation, may be obtained from the Computational Memory Lab's Web site (http://memory.psych.upenn.edu).

PyParse must be called from the command line and accepts values for two arguments: the sound file to be scored (parsed) and a text file listing candidate words to be identified (the *word pool*). Although one could pass a very large word pool containing virtually any possible response, it is often satisfactory to define the word pool as the list of words used in the experiment. For many of the studies in our own laboratory, we use the nouns found in the Toronto Word Pool (Friendly, Franklin, Hoffman, & Rubin, 1982). PyParse accepts a number of options via the command line to identify properties of the sound file, including the sampling rate, the number of channels (1 = mono or 2 = stereo), the background noise profile, and the file format (for a full list, see Table 1). In most cases (when using standard .wav files), PyParse automatically detects the values of these properties. PyParse can also be given multiple sound

Table 1
Command-Line Options Recognized by PyParse

-a, –raw =	Read in raw sound data.
-c, –channels =	Number of channels in recording (mono or stereo).
-b, –bandpass =	Band-pass filter range (e.g., -b1000,16000).
-e, –bigendian =	Set sound data endianness to big.
-d, –diffmode =	Display difference between channels in stereo sound file(s).
-f, –format =	Sample width in bits. Possible values: 8, 16, 24, 32.
-h, –help =	Show usage info.
-o, –onsets =	Automatically guess sound onsets.
-n, –noise =	Path to a .wav file with a recording of typical background noise. Useful for more accurate onset detection.
-r, –rate =	Sampling rate of sound files.
-w, –wordpool =	Wordpool file.
-z, –zerobased =	Start wordpool indexing from zero.

files at once (e.g., one can easily reference all of the sound files stored in a particular directory), in which case it then conveniently presents them for scoring one at a time.

Sample Run

We describe the basic parsing procedure by way of example. All of the keystrokes corresponding to the commands referenced in this example are listed in Table 2. Consider again a basic free recall experiment. A participant is presented with a short list of words and is instructed to recall the list in any order following the last word presentation. Words are randomly drawn from the Toronto Word Pool (Friendly et al., 1982), which is stored (with one word per line) in a file named wordpool.txt. The experiment is controlled using a software package that supports digital voice recording and can present text at accurately timed intervals. The study–test procedure is repeated for several lists, and the recall period for each is stored in a file whose name corresponds to the list index (0.wav, 1.wav, etc.). For convenience (though optional), the words that make up each list are stored in a parallel set of files (0.lst, 1.lst, etc.), again with one word per line.

In order to begin parsing the first file, we issue the following command: `pyparse -w wordpool.txt 0.wav`. This tells PyParse that we want to score the file 0.wav and that valid responses were drawn from the list of words found in wordpool.txt. PyParse filters the given sound file with a band-pass range of 1000–16000 Hz (although the range can be changed using a command-line option, we have found this default value to work well for isolating human speech) and shows the resulting waveform on a screen similar to the one in Figure 1. The user can modify

Table 2
Commands That PyParse Accepts and Their Default Keyboard Mappings

Playback	Space bar	Starts and stops playback.
	Ctrl + Z	Replays the last 200 msec prior to the cursor's current position.
	Ctrl + X	Decreases playback speed.
	Ctrl + C	Increases playback speed.
	Ctrl + V	Resets playback speed to normal.
Cursor	left arrow	Moves cursor to the left.
	right arrow	Moves cursor to the right.
	When the above two commands are used in conjunction with the Ctrl key, the step size is larger. When used in conjunction with both the Ctrl and Shift keys simultaneously, the step size is larger still (1,000 msec).	
	Ctrl + /	Centers the screen on the cursor's current position.
Anchoring	Ctrl + A (first time)	Sets the first anchor point.
	Ctrl + A (second time)	Sets the second anchor point and enters anchor mode.
	Ctrl + A (third time)	Exits anchor mode.
	left arrow	In anchor mode, moves the second anchor point left.
	right arrow	In anchor mode, moves the second anchor point right.
	Ctrl + left arrow	In anchor mode, moves the first anchor point left.
	Ctrl + right arrow	In anchor mode, moves the first anchor point right.
Scoring	(A–Z)	Types in the response box to narrow the list of candidate words from the word pool.
	Tab	Autocompletes the response box with the first word in the list matching what's been typed so far.
	Enter	Enters the selected word at the cursor's current position.
	Ctrl + Shift + I	Enters an intrusion at the cursor's current position.
	Ctrl + Delete	Deletes the current word marker.
	Ctrl + M	Moves the word marker that was last selected to the cursor's current position.
Magnification	Ctrl + up arrow	Zooms in on the *y*-axis (amplitude).
	Ctrl + down arrow	Zooms out on the *y*-axis (amplitude).
	Ctrl + .	Zooms in on the *x*-axis (time).
	Ctrl + ,	Zooms out on the *x*-axis (time).
	If the above four zoom commands are used in conjunction with the Shift key, the step size is larger.	

Note—The key mapping for each command can be changed in a configuration file.

the level of magnification on both the *y*-axis (affecting the amplitude of the display) and the *x*-axis (affecting how much of the waveform is shown on the screen at any one time) with a single keystroke. The bottom half of the screen contains, from left to right, the response box and word pool, a list of the vocalizations marked so far, along with their corresponding onsets, a volume slider, a playback speed display, and command buttons for closing the application in one of two ways (both of which are explained later).

We begin scoring the file by listening to the first vocalization shown on the screen (see, e.g., Figure 1), using the space bar to start and stop playback. The left arrow key is then used (possibly in conjunction with one or more modifiers to traverse a larger distance; see Table 2) to reposition the cursor prior to the start of the vocalization, allowing for ample slack room.

Locating the vocalization's onset. The best estimate of the vocalization's onset can be found in one of two ways. The first method involves incrementally moving the cursor using the right arrow key (without any modifiers, this is the smallest step size and corresponds to a 5-msec default). After each step, Ctrl+Z is used to play back the last 200 msec of the file and gauge whether the vocalization has started. Although this method works well for most vocalizations, a more flexible method is sometimes required, especially when marking words that start with soft fricatives. This second method involves dropping two anchor points to restrict playback to a precise area of the waveform. The first anchor point is dropped (Ctrl+A by default) prior to the hypothesized onset of the vocalization, and a second anchor point is dropped to the right of the first by pressing Ctrl+A again after repositioning the cursor. In anchor mode, the left and right arrow keys are used to move the right anchor point and, in conjunction with the Ctrl key, the left anchor point. Pressing the space bar plays only the part of the file that falls between the two anchor points. This allows the user to define an arbitrary window and shift it in small increments until a precise estimate of the onset is found.

Labeling vocalizations. Once we obtain the best estimate of the onset using one of the two methods described above, we type the word that was previously played back. As more and more of the word's prefix is typed into the response box, the word list is filtered to display only the matching words. If a corresponding (optional) .lst file is found as described above, the words contained in the file appear at the top of the word list in bold. This allows the user to more easily identify a mumbled word if it resembles a word that was on the list being scored. Once enough characters of a word are typed to uniquely identify it, the Tab key can be used to auto-complete the word in the response box. Finally, the Enter key is used to place the word at the onset's location.

Marking intrusions. If the vocal response corresponds to a word that does not appear in the word list, it is marked as an intrusion by pressing Ctrl+Shift+I, instead of the Enter key. If the response was nonsensical or consists of the participant talking to him- or herself or to the person running the experiment, it is marked in a special way by first typing "VV" in the response box and then pressing Ctrl+Shift+I, as if marking an intrusion. In our laboratory, if such vocalizations last more than 1 sec, we also mark them at each 1-sec interval. These time stamps can then be used to identify the corresponding neural data (if measured during the experiment) so that they can be treated with caution or discarded altogether.

Output file. After labeling a vocalization, PyParse writes a line entry to a temporary file (this file has the same base name as the sound file being scored and the extension .tpa) with the following three columns of information: the onset of the vocalization in milliseconds, the index of the word in the word pool file (−1 for intrusions), and the word itself.

Finishing the session. After labeling the first vocalization, we proceed to scoring the remainder of the file in a similar fashion. One can quit PyParse in one of two ways, depending on whether the entire file has been scored. If the entire file has not been scored, clicking the "Quit" command button will close PyParse while leaving the temporary .tpa file in place. Relaunching PyParse with the same sound file will automatically load the information stored in the .tpa file and allow the user to pick up where he or she left off. Once the entire file is traversed at least once, the "Done" button becomes available. Clicking it changes the extension of the .tpa file to .par, signifying that this sound file has been scored. The resulting .par file can then be processed with the programming language of choice.

Automatic Onset Detection

We have put considerable effort into optimizing the accuracy, consistency, and speed with which recordings can be manually scored (cf. the Usage Statistics section). In general, scoring audio recordings consists of two steps: finding the onset of the vocalization and labeling the vocalization. Recordings made in a laboratory setting are usually of very high quality, and one can label a vocalization rather quickly. However, locating accurate and consistent onsets is difficult even in a recording with a high signal-to-noise ratio. Automating this task thus has the potential to save a great deal of time.

While recent advances in automatic endpoint detection have focused on algorithms that improve accuracy in high-noise environments, an algorithm that remains popular for use in low-noise environments is that of Rabiner and Sambur (1975). We have implemented their algorithm in a PyParse add-on called PyWR (Python word recognition).

The onset detection feature can be invoked in two ways: via the −o command line option given to PyParse (for a list of all command-line options, see Table 1), and via a stand-alone program capable of processing multiple files at once. When the −o option is chosen, PyParse first checks whether a previous session is saved in a corresponding .tpa file. If so, the −o option is ignored, and the previous session is restored. If a previous session is not found, PyParse runs the onset recognition algorithm on the given file. Each onset is labeled with a question mark, signifying that it has yet to be labeled. If the recordings were not made in a noise-free environment, the −n or −bgFile options are used to point PyWR to an audio file containing a 1-sec recording of typical background noise. PyWR uses this file to tweak its parameters. Note, however, that the algorithm still assumes that the overall signal-to-noise ratio is high and that whatever little background noise exists is stationary.

The second way to invoke the onset detection feature is via the stand-alone program `pywr_onsets.py`: `pywr_onsets.py file1.wav [file2.wav file3.wav . . .]`. A .tpa file with the detected onsets is generated for each .wav file given to the program. These files can later be loaded into PyParse so that the onsets can be double-checked and labeled. If the recordings were not made in a noise-free environment, the `-bgFile` option can be used to specify a background noise profile as was described above. This batch mode feature, which allows multiple files to be marked without user interaction, can save a lot of time when scoring a large number of files.

Although conceptually simple, the algorithm of Rabiner and Sambur (1975) accurately identifies onsets for a large percentage of vocalizations. It does, however, have two limitations and represents only a first step in automating onset detection in PyParse. First, as it was described in its original form, the algorithm expects only one vocalization within the recording. Although this restriction has been lifted in our implementation, the modified algorithm does a poor job of separating words that are spoken in rapid succession. In such cases, the algorithm treats the entire segment as one vocalization and only marks the onset of the very first word.

The second limitation of the algorithm is that it often overshoots the onsets of words that begin with weak fricatives (e.g., /f/), because their energies ramp up slowly. Although Rabiner and Sambur (1975) addressed this shortcoming with a secondary refinement phase, it does not work as well as would be expected when vocalizations are made in relatively rapid succession and stored within a single file (see the Implementation section below). The user must be mindful of both of these shortcomings and manually mark the onsets that the algorithm misses.

In recognition experiments run in our laboratory, where responses are limited to two words (e.g., "yes" and "no"), we modify the responses so that they start with the same sound. For instance, instead of saying "yes" or "no," participants say "pes" or "po." Since the energy of the /p/ sound ramps up quickly, the algorithm is very good at accurately locating the onsets of these words. Also, since both start with the same sound, onset estimates are consistent across words. Such a feature is important when looking at response time data.[1]

Word Recognition

Automatic word recognition is available for use with experiments in which the pool of possible responses is relatively small. For example, we have successfully automated scoring data from a recognition experiment in which the pool of responses consists of the words "pes" and "po" (for "yes" and "no"). This feature is part of the PyWR add-on and is accessible via a command-line interface to facilitate the ability to score data in batch mode. Using this feature involves two steps: training a classifier on labeled data collected during a pre-experimental training phase and pointing the classifier to unlabeled data collected during the experiment.

For the training step, PyWR expects as input one audio file for each valid word. The file must contain a record-

ing of the word being repeated multiple times, with a short pause between repetitions. The stand-alone program `pywr_train.py` is used to train the classifier and takes as its single argument the directory containing the audio files to use: `pywr_train.py [train_dir]`. PyWR looks at each .wav file in the directory and creates a corresponding .hmm file with the model parameters to use for classification.

The stand-alone program `pywr_classify.py` is used to classify a set of unlabeled audio files: `pywr_classify.py wordpool_file model_dir file1.wav [file2.wav file3.wav ..]`. Here, `wordpool_file` is the path to the word pool file used during the experiment (i.e., the file that would be passed to PyParse if the data were classified manually), `model_dir` is the directory containing the .hmm files generated during the training phase, and the rest of the arguments are the audio files to classify. A .tpa file is generated for each .wav file, which can then be loaded into PyParse to quickly check the accuracy of the labels.

In our laboratory, we collect training data for each participant and use it to train a unique classifier for their voice. Although this may be excessive for classifying words from a very small word pool, it allows us to adapt the classifier to individual differences and achieve very high recognition accuracy.

As a final note, the current implementation assumes that participants are not trying to trick the system with invalid responses. This assumption is reasonable, since the data in most such cases should probably be discarded.

IMPLEMENTATION

The front-end interface and most of the higher level features were written in Python, making use of `wxPython` for the GUI, and `SciPy` and `NumPy` for postprocessing the audio data. At a lower level, audio data are processed by a thin wrapper around `RtAudio`, `libsndfile`, `libsamprate`, and `SoundTouch`, written in C++ and made available to Python using SWIG.

We forgo a detailed discussion of many implementation details, since they are not in themselves novel. The source code is available on the Computational Memory Lab's Web site (http://memory.psych.upenn.edu) and is accessible to anyone with Python programming experience. We do, however, discuss the details surrounding the current endpoint detection and word recognition algorithms. Not only is their application to scoring psychological data novel, but they represent the features of PyParse that we believe can use the most improvement.

The Automatic Endpoint Detection Algorithm

We describe a slightly modified version of the endpoint detection algorithm of Rabiner and Sambur (1975). For further details, we refer the reader to the original article.

Both automatic and manual (human) endpoint detection is an especially challenging problem in the presence of background noise. In the case of automatic detection, successfully separating speech from background noise requires a sophisticated filtering scheme and a detailed

statistical model of the signal. Since recording conditions in the laboratory are under the experimenter's complete control, we assume that background noise is minimal and that the signal-to-noise ratio is very high.

A 10-msec window is first swept across the signal while the energy of each frame is noted. By *energy* here, we simply mean the sum of the magnitudes of all of the samples that fall within the boundaries of the window. An analogous operation is performed on 100 msec of background noise, recorded in the testing room. Two statistics are computed, based on the peak energy of the input signal (IMX) and the mean energy of the background signal (IMN):

$$ITL = \min(0.03 \cdot (IMX - IMN) + IMN, 4 \cdot IMN) \quad (1)$$

$$ITU = 5 \cdot ITL \quad (2)$$

Respectively, these two values represent the lower and upper thresholds used to segment voiced parts of the recording, as is described below.

A 10-msec window is then again swept across the signal. If a frame whose energy exceeds the lower threshold (ITL) is found, the center of the frame is marked as a potential onset. The window is further swept across the signal until one of four things occurs. If a frame whose energy exceeds the upper threshold (ITU) is found or if the signal's energy is maintained above the lower threshold for a predetermined amount of time (200 msec by default), the previously marked sample is confirmed to be an onset. If a frame whose energy falls below the lower threshold is found before meeting either of these two conditions, the hypothesized onset is discarded. This filters out artifacts that arise from false starts. A hypothesized onset is also discarded upon reaching the end of the signal.

After locating an onset, a window continues to be swept across the signal looking for the corresponding offset. Although we are not explicitly interested in knowing where a vocalization ends, we locate the offset for two reasons. First, since the signal can contain multiple vocalizations, we know to start looking for the next vocalization in the frame following the offset. Second, if the length of a vocalization falls below a configurable threshold (100 msec by default), the vocalization is usually too short to be of value and the endpoints are discarded.

Rabiner and Sambur (1975) described a refinement phase meant to correct the onsets of words that begin with weak fricatives. A 10-msec window is swept across the 250-msec segment of the recording preceding the first onset estimate (for offsets, they look at the next 250-msec segment) while the number of zero-crossings in each frame is calculated. A large number of zero-crossings is taken as evidence of a vocalization. If the number of zero-crossings in three or more frames is further than two standard deviations away from the mean number of zero-crossings in a typical frame of silence, the onset (or offset) is adjusted to be at the center of the earliest (or latest) frame exceeding this threshold.

This approach is reported to work well in the domain addressed by Rabiner and Sambur (1975), where they expected recordings to contain a single vocalization. However, we have found mixed results in practice when using this approach for recordings made in a laboratory setting and containing multiple vocalizations per file. Although it provides an accurate correction in some cases, in others it positions the onset estimate prior to where a human operator would. This problem occurred often enough during testing to warrant turning the refinement step off by default.

The Automatic Word Recognition Algorithm

The current implementation of the word recognition feature can automatically score data from experiments in which the pool of possible responses is relatively small. In particular, we have successfully used it to score data from a recognition experiment with a response pool of two words ("pes" and "po"). Here, we briefly describe the current implementation, based largely on the work of Rabiner, Juang, Levinson, and Sondhi (1985). The modular form of PyParse and PyWR allows one to easily drop in a more sophisticated algorithm capable of recognizing words from a larger lexicon at a later date.

Training data are obtained from each participant by having them repeat each of the possible responses several times. In our recording environment, we can achieve an average classification rate (across participants) of over 99% using a training set consisting of 30 repetitions of each word.

The training data are band-pass filtered between 1000 and 16000 Hz to remove noise, and a pre-emphasis filter is applied to boost the attenuated energy that is typical at higher frequencies of human speech. The endpoint detection algorithm outlined above is used to identify the voiced portions of the signal. Mel-frequency cepstral coefficients, which have been employed in speech recognition for some time (Davis & Mermelstein, 1980), are computed for a moving window of the voiced segment of the signal. They constitute the features used for classification.

A hidden Markov model in which the observations in each state are modeled as a mixture of Gaussians is fit separately to each word. Parameter estimates are obtained using the standard Baum–Welch algorithm, an iterative procedure that finds estimates that maximize the likelihood of the training data (e.g., Rabiner, 1989).

Unlabeled data are preprocessed in exactly the same way as the training data. To label a vocalization, the Viterbi algorithm is first used to find the most likely sequence of state transitions under each model. From these, the most likely overall sequence is selected, and the label associated with the corresponding model is used.

USAGE STATISTICS

Onset Consistency

PyParse allows multiple people to score different portions of the same data set without sacrificing the consistency with which onsets are marked. Five research assistants who use PyParse on a regular basis scored the same set of three recall periods from a random participant of a large free recall experiment. Together, the three recall periods contained 43 responses. The mean (across responses)

of the standard deviation between research assistants was 12 msec (± 2 msec). The mean deviation between two of the most experienced users was 3 msec (± 0.5 msec).

Efficiency

A typical parsing session was timed for each of five research assistants. On average, it took less than 30 sec to listen to a vocalization, to rewind the cursor and use one of the methods described in the Usage section to find the best estimate of the onset, and to label the vocalization.

Word Recognition Accuracy

A recognition experiment in which the pool of valid responses was limited to two words ("pes" and "po") was scored both manually and using the automatic word recognition feature. The data set contained 24 participants with an average of 1,475 responses each. The mean classification accuracy across participants was over 99.4%, whereas the worst accuracy for any 1 participant was 97.1%.

FUTURE ENHANCEMENT

Although the current version of PyParse has enabled us to efficiently collect interresponse time and output order data in numerous studies, there are a number of major limitations that should be addressed in future work. The most significant limitation is the need to manually identify spoken words in data collected from most studies. With advances in computer technology and speech recognition algorithms, it should be possible to accurately identify a large portion of words, letting the user identify only those words that the speech recognition algorithm could not identify with high confidence.

Another limitation is the unreliable nature of the onset detection algorithm under the conditions previously described. We have experimented with a number of primitive algorithms, none of which were completely satisfactory (especially in cases where words were slurred together). By using a more sophisticated word model, it should be possible to automatically detect voice onsets with a higher degree of accuracy. As with word recognition, one could envision an algorithm that gauges its own confidence, allowing the user to manually identify onsets for troublesome words. A word model incorporating both acoustic and semantic information could potentially solve both of these problems simultaneously.

The possibility of automatic parsing raises the prospect of providing real-time performance feedback. Several scenarios come to mind. First, in neuropsychological assessment procedures, one could have the computer automatically adjust the difficulty of the lists on the basis of patient performance. Second, in studies of learning, the computer could repeat study–test trials until some performance level is achieved. Third, by dynamically monitoring interresponse times, one could adjust the recall period depending on the amount of time that has elapsed since the last response. The possibility of triggering experimental events as a function

of the sequence of recalled items could open a wide range of new areas of study in verbal recall. Finally, because recall tasks are playing an increasingly important role in detecting memory impairments associated with neurological disease, automatic data collection and scoring can become an important part of remote (e.g., phone-based) monitoring systems. PyParse is only a first step toward these far more ambitious aims, but it highlights the richness of data that can be gleaned from recall experiments and will hopefully stimulate the development of more sophisticated tools for scoring verbal recall protocols.

REFERENCES

Davis, S., & Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, & Signal Processing*, **28**, 357-366.

Friendly, M., Franklin, P. E., Hoffman, D., & Rubin, D. C. (1982). The Toronto Word Pool: Norms for imagery, concreteness, orthographic variables, and grammatical usage for 1,080 words. *Behavior Research Methods & Instrumentation*, **14**, 375-399.

Geller, A. S., Schleifer, I. K., Sederberg, P. B., Jacobs, J., & Kahana, M. J. (2007). PyEPL: A cross-platform experiment-programming library. *Behavior Research Methods*, **39**, 950-958.

Kahana, M. J. (1996). Associative retrieval processes in free recall. *Memory & Cognition*, **24**, 103-109.

Murdock, B. B., & Okada, R. (1970). Interresponse times in single-trial free recall. *Journal of Experimental Psychology*, **86**, 263-267.

Patterson, K. E., Meltzer, R. H., & Mandler, G. (1971). Interresponse times in categorized free recall. *Journal of Verbal Learning & Verbal Behavior*, **10**, 417-426.

Pollio, H. R., Richards, S., & Lucas, R. (1969). Temporal properties of category recall. *Journal of Verbal Learning & Verbal Behavior*, **8**, 529-536.

Polyn, S. M., Norman, K. A., & Kahana, M. J. (2009). A context maintenance and retrieval model of organizational processes in free recall. *Psychological Review*, **116**, 129-156.

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, **77**, 257-286.

Rabiner, L., Juang, B.-H., Levinson, S., & Sondhi, M. (1985). Recognition of isolated digits using hidden Markov models with continuous mixture densities. *AT&T Technical Journal*, **64**, 1211-1234.

Rabiner, L., & Sambur, M. (1975). An algorithm for determining the endpoints of isolated utterances. *Bell System Technical Journal*, **54**, 297-315.

Rohrer, D., & Wixted, J. T. (1994). An analysis of latency and interresponse time in free recall. *Memory & Cognition*, **22**, 511-524.

Wingfield, A., Lindfield, K. C., & Kahana, M. J. (1998). Adult age differences in the temporal characteristics of category free recall. *Psychology & Aging*, **13**, 256-266.

NOTE

1. We thank Professor Saul Sternberg for suggesting the "pes"/"po" variant described above.