

MUSE: AN OPEN SOURCE SPEECH TECHNOLOGY RESEARCH PLATFORM

Peter Cahill and Julie Carson-Berndsen

CNGL, School of Computer Science and Informatics,
University College Dublin, Dublin, Ireland.

{peter.cahill|julie.berndsen}@ucd.ie

ABSTRACT

This paper introduces the open source muster speech engine (Muse) for speech technology research. The Muse platform abstracts common data types and software as used by speech technology researchers. It is designed to assist researchers in making repeatable experiments that are not hard coded to a specific platform, language, algorithm, or corpus. It contains a script language and a shell where users can interact with various components. The presentation of this paper will be accompanied by a demo at the SLT workshop.

Index Terms— Muse, software, open source

1. INTRODUCTION

Muse was developed to assist the interaction of existing speech software tools, where components may represent several existing systems and Muse is essentially the layer that makes the components function together. In some cases, this may simply be converting between data file types, in other cases it may involve generating appropriate configuration files, converting data types, training acoustic models and exporting data to some external analysis software.

In its current state, programming skills are required to use Muse, where the existing components can help reduce the development overhead in new experiments. For example, consider a user that wishes to experiment with a specific data set, where the data set is input, processed by 2 external applications, and then output to MATLAB for statistical analysis. Much of the required functionality may already exist in Muse, allowing the researcher to spend less time developing software. In the example, it could be that each application in the process is relatively trivial to use but perhaps each application needs data in different formats. Muse can automate the data format conversion where necessary and ensure that the relevant applications are configured correctly.

The architecture of Muse was carefully designed to avoid issues such as hard coding to a specific machine or platform, so that users can share their added functionality with other users who work on different configurations and systems. In addition to this, algorithms are automatically discovered at

runtime enabling them to propagate to all tools and experiments automatically. For example, many experiments involve reading speech annotation files of some form. A user may need to add support for a new annotation format, where, when a new format is added, all programs and experiments that use Muse to read annotations will automatically support the new format without any additional changes or updates being required.

In order to encourage the development of new components, benefit the speech community, and enable the continued evolution of the platform, the Muse platform was recently released as open source under the Apache 2.0 license (available with documentation from: <http://muster.ucd.ie/muse>). This license enables the platform to be used for both academic and commercial purposes, similar to other permissive open source licenses.

This paper, which coincides with the release of the Muse platform, explains the underlying design concepts, features and roadmap of the project. The remainder of this paper is structured as follows, section 2 describes the current release of Muse. Section 3 describes the architecture of Muse, section 4 details the main components in the current version and section 5 provides a use case scenario of the platform. Section 6 discusses issues and the future roadmap for Muse. Section 7 concludes.

2. MUSE

The current version of Muse contains a collection of components which have been developed as part of speech technology research in the Muster lab at University College Dublin. While the current release will not have all components necessary for many speech research experiments, it has a significant number of components so that it can be quicker to add a missing component than to develop an experiment from scratch.

Much of the current functionality in Muse is focused on abstracting core technologies from existing speech toolkits. It is because of this that many of the existing components support the file formats of external tools and contain wrapper components for invoking external tools. One example of this is that hidden Markov models (HMMs) can be trained in Muse via HTK [1] or Sphinx [2], where the user only needs

to change 1 line in a script in order to alternate between each toolkit.

A significant amount of the work to date on Muse has been focused on creating an architecture that makes it easier to interchange parts of an experiment. In some cases it can be appropriate to test on more than one data set or to interchange various algorithms. The following section describes the fundamental concepts that are part of the Muse architecture.

3. ARCHITECTURE

In the design of any large system, the architecture is the most critical part of the development process. The architecture of the Muse platform was carefully designed to accommodate researchers. In particular, several points of consideration include:

1. Researchers will often not have the same software development skills as professional developers.
2. The Muse platform must reduce the time overhead in experiment development.
3. It must be trivial to develop experiments that are independent of any particular operating system and avoid hard coding.
4. It should be very easy to change part of an experiment (for example, format of data set, size of data set).

This section describes some key aspects of the architecture of the platform.

3.1. Platform Overview

The architecture of the Muse platform consists of 3 levels: core (L1), libraries (L2), and scripts (L3). Figure 1 illustrates the 3 levels with some example elements at each level. For illustration purposes many of the existing elements were omitted.

1. The core of the platform contains the essential core elements of Muse. These are reusable building blocks that are used by all 3 levels. Examples elements in the core include basic classes (matrix, table, signal, annotation, etc.) and abstract machine learning classes. The core of the platform is compiled as a standard library, consisting of components which are unlikely to change and have reached a level of maturity.
2. Libraries are Muse scripts (text files) that are installed in a specific *muse/lib* directory. All methods and data types in these scripts are automatically made accessible to the Muse platform. Library scripts can be edited as text files, and will be automatically reloaded by Muse without the need for compilation and installation steps.

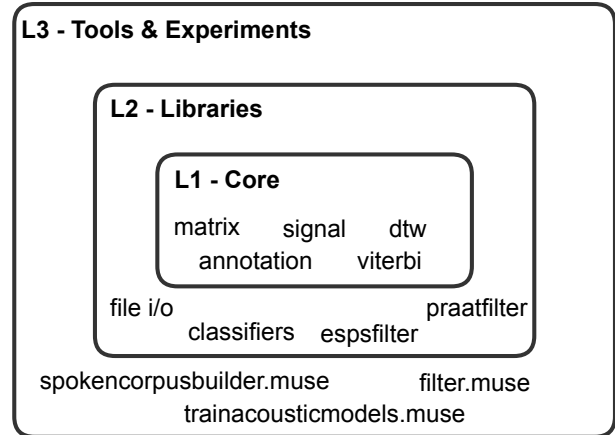


Fig. 1. High-level overview of the Muse architecture.

For example, consider that a user develops a library script that contains support to read a new annotation file format. By placing the library script into the *muse/lib* directory, all experiments and tools using Muse will automatically support the new file format without any changes or updates being required.

3. Scripts are the same format as the library Muse scripts, but their methods and data types are internal and not accessible to other scripts. Scripts appear as executable programs (similar to a typical bash shell script).

For a typical user, an experiment would be a script (level 3). Their script may use some file input output support that is present in a library (level 2), and then export their results (datatypes level 1) in a MATLAB file as supported by a level 2 library.

3.2. Choice of Programming Language

To maximise the flexibility of the platform, it is important that new users can use components for tasks that they were not specifically designed for. In order to ensure that such flexibility is achieved, the Muse platform is accessed by an application programming interface (API). While C and C++ are the programming language of choice for many expert programmers, many less experienced developers much prefer either Java or C#, as these are more common amongst modern university courses. The advantage of C/C++ is that software can be efficient in terms of processing and memory resources if it is well designed and optimised, but it takes a significant amount of additional development time to ensure that the developed software is efficient. Both Java and C# have the advantage that they have less compilers available, resulting in their syntax and features being somewhat more standard, as well as features like memory management are included. The disadvantage of these languages is that they often consume more memory than efficiently implemented C/C++ software.

The deciding factor for the Muse platform was that C# has an open source compiler implementation (Mono project [3]), where programs can run across many platforms (Windows, Linux, Mac OS X, FreeBSD, iPhone, Android). C# also has several language features that made it the appropriate choice for the Muse platform: generics, events, delegates (anonymous methods) and lambda expressions. In addition to this, it is possible to automatically convert any Java library to a C# library using the IKVM toolkit (distributed as part of Mono).

One additional benefit of using C#, is that any experiments or tools can be made available through a web server without much additional effort. Tools and experiments written in C# can be imported into a standard web server that supports ASP.NET (for example, Internet Information Services (IIS) on Windows or Apache/Mono on Linux).

3.3. Prototyping - the Muse shell

When using traditional compiled programming languages (such as C/C++), it is necessary to perform development in stages, where a feature is added, then the system is rebuilt, reinstalled and then tested. Interpreters can be used to streamline some of this process, where a program is run line by line rather than in bulk. While this can make program execution slower, it does offer the advantage that the user can write the program as it is running. If there is an error in something the user entered, they are notified immediately after they entered it.

The Muse platform includes a shell, which is somewhat comparable to the well known bash shell on Unix machines. The main difference in the Muse shell is that its syntax is C#, where it supports the full C# language (specification v4.0) and makes all Muse components completely accessible to the user. This enables the user to interact with components in a real time way that is not possible in conventional compiled languages.

The Muse shell supports both Muse scripts and interactive user input. When running Muse scripts, they are compiled and then run. This helps speed up experiment execution times, while still allowing the user to enter commands via the interpreter during development.

3.4. Scalability

For many experiments, the computing power of modern computers is sufficient. However, if an experiment involves a real world data set and machine learning, resources can be an issue. As the Muse platform was designed to wrap around existing speech tools, APIs are included to run external software programs. These APIs are implemented as a queue, where several processes may be queued, and the system will wait for results when necessary.

The back end of the queue APIs can be easily changed, so that on an older computer, only 1 process is run at a time,

on a modern desktop, it may be reasonable to run 4 or more process at a time. Similarly, the API can be configured for the Muse platform to utilise a computing cluster, where jobs will be automatically deployed to a cluster and Muse will be aware of when the appropriate jobs are complete. Switching between using a single process, multiple processes, or a computational cluster is configured in a single line of the Muse configuration file, and it affects all experiments and tools that use Muse.

3.5. Basic data types

The standard installation of the Muse platform supports a wide range of data formats. For basic (numeric) data input / output, it supports: MATLAB, HTK, wavesurfer [4], LRN, raw. Supported common speech annotation file formats include: BOSS [5], TIMIT, MLF, EmuLab, Sphinx, Praat, CTM and TRN. In addition to this, there are readers for several common speech corpora: Voxforge, Switchboard, Romanian Babel, Roger (CSTR), BOSS, BREF, FestVox, Kiel, TIMIT.

4. MAIN COMPONENTS

This section describes some of the main components in the current version of Muse: spoken corpus, filter, acoustic models and machine learning. These components represent elements at 1 or more of the 3 levels in the Muse architecture. For example, the spoken corpus component is represented as a datatype in L1, corpora are imported via the *spokencorpus-builder.muse* script at L3, which in turn uses the annotation and corpora file readers from L2. These components were developed as they are quite generic and can be useful in many speech technology experiments.

4.1. Spoken corpus

In Muse, the term spoken corpus is used to refer to a speech data set. These typically consist of a collection of utterances with orthographic transcriptions. A spoken corpus may also have any number of annotation layers and can have numeric parameters (for example, acoustic parameters).

The motivations for the spoken corpus component was to make it trivial to change data sets used in an experiment. This was specifically done so that experiments could be easily repeated on different languages without increasing the workload on the user. All of the corpus processing steps in Muse use the spoken corpus component rather than processing individual audio files as it allows tasks to be processed in a batch mode. This results in corpora being processed efficiently as batch processing will be used where possible.

The main tool for spoken corpora is the *spokencorpus-builder.muse* script. This script imports any external corpora,

and saves them into a single file format. Currently, the following corpus formats are supported: Voxforge, Switchboard, Romanian Babel, Roger (CSTR), BOSS, BREF, FestVox, Kiel, TIMIT. Support for new corpora is quite trivial to add, where most of the existing formats require approximately 10-20 lines in a Muse library script.

After any corpus is imported via the `spokencorpus-builder.muse` script, several other scripts can be used for post processing. Other corpus scripts offer additional functions, such as: merging corpora, edit corpora annotations (HTK HLEd files are supported) and calculate statistical information on annotations.

4.2. Filters

In the context of Muse, ‘filter’ is a generic term that is used to represent any form of numeric calculation from a speech signal. For example, the term filter is used to refer to represent signal transformations and the calculation of acoustic parameters. As such filters are commonly used in speech technology research, Muse contains a flexible `filter.muse` script that allows the user to apply available filters to either audio files or to entire corpora.

The `filter.muse` script allows the user to specify a list of filters which can be applied in a single processing step. For example, it is possible to extract 12 MFCC parameters via HTK, calculate their delta and acceleration coefficients, estimate pitch parameters using the Entropic Signal Processing System (ESPS) [6], and then calculate intensity parameters using Praat [7] - all in one single command line. Such a command could be used for a single audio file, or for an entire corpus, where for each file in the corpus, the parameters are joined into a single matrix per audio file, such that there would be 38 ($MFCC_{DA}=12*3$, $pitch=1$, $intensity=1$) parameters for each 10ms window of speech in the example presented. Additional functionality in the `filter.muse` script enables the user to specify post processing for any of the calculated filter parameters, where, delta and acceleration coefficients, log, or normalisation parameters can be calculated. Similarly, linear and cubic spline interpolation can be applied to filter parameters.

In the current version of the Muse platform, several filters are included: HTK, Praat (pitch estimation via auto correlation or cross correlation, and intensity calculation) and ESPS (estimation of formants and pitch). Each filter type is implemented as a Muse library script, where new filters can be added by implementing a library script that provides the necessary functionality and placing it in the `muse/lib` folder.

4.3. Statistical acoustic models

In many domains of speech technology, acoustic models are used to process speech signals. Statistical acoustic models are commonly hidden Markov models (HMMs), where they

often represent the statistical information of the acoustics of phones, sometimes within a specific context.

In Muse, a `trainacousticmodels.muse` script is included which can be used to train a set of models from an annotated corpus. The user can specify model parameters, such as window size or overlap, feature parameters (as created by the `filter.muse` script), number of HMM states, etc. Similarly the user can specify which method they wish to use to create the models. Currently available methods include Sphinxtrain (to train models for sphinx), HTK Baum-Welch method (for temporally annotated corpora), or HTK embedded Baum-Welch method (which can be used for corpora without temporal annotations).

Regardless of the method chosen, Muse is able to create models from any annotated corpus. When training the models, Muse generates all necessary configuration files needed and completely abstracts the user from the underlying toolkit being used to train the models.

As described in the scalability section (section 3.4), processes like the acoustic model training procedure use the Muse process queue. This results in some training methods training models in parallel, or as in the case of the embedded Baum-Welch method, the dataset will be divided and then accumulated to speed up the training process. If the user’s installation of Muse is configured to use 4 processes, the training process will utilise all 4 processes in order to train models quicker, or similarly, if Muse is configured to utilise a computational cluster, all models will be automatically trained on a cluster, without the user needing to make job scripts or monitor progress.

Similar to the filter methods, additional acoustic model training methods may be added as Muse libraries.

4.4. Machine Learning

Although machine learning is a commonly used tool amongst speech technology researchers, the current version of the Muse platform does not contain any internal implementations of machine learning algorithms. Instead, Muse contains abstract machine learning classes, which allow for external machine learning toolkits to be used. For example, Muse contains a generic classifier class, where several external toolkits may be used to provide the actual classifier (for example, decision trees, neural networks, etc.). This makes it trivial to change which machine learning algorithm is being used by an experiment.

The current version of Muse incorporates a complete version of Weka [8]. Weka is a data mining and machine learning toolkit, which includes support for many classifier types (several types of decision trees, artificial neural networks, etc.). This enables users to train models either from within Muse, or through the Weka user interface, where Weka provides the user with information on the statistical distribution of the data set, as well as it enables the user to evaluate which classifi-

cation algorithms are best for a task. In addition to Weka, a support vector machine library, svmlight, is also supported.

5. USE CASE EXAMPLE

In order to demonstrate Muse, an example scenario is presented. Consider that, as part of an experiment, a user needs to train context independent HMM phone models that model formants, pitch, and intensity. While there are many tools available that can estimate these parameters, most of the existing tools output parameters to different file formats. Similarly, HMM toolkits use different file formats and configurations, and any corpus data is likely to be in its own format as well. In this example, the tools considered are ESPS for formats and pitch, and Praat for intensity. HTK is used to train the HMM models.

In a traditional experiment set up, the user would need to:

1. Convert corpus speech data into appropriate file formats for the parameter estimation tools.
2. Create scripts for each of the parameter estimation tools to be applied to each audio file in the corpus.
3. Combine the results from the parameter estimation tools, and store them in the HTK parameter file format.
4. Create all the necessary configuration and setup files for HTK.
5. Convert the corpus' annotations to the HTK label format.
6. Invoke the appropriate tools within HTK to create the models appropriately.

A significant amount of work in this example involves converting between data types, making scripts to parse the corpus data, manually creating configuration files and scripts that can train the models.

Figure 2 illustrates how this process can be done using Muse. There are 3 steps in the task:

1. Import a corpus from an external format.
2. Apply the Muse filter script to the corpus to calculate the parameters for the HMMs.
3. Run the *trainacousticmodels.muse* script to train the models.

In this process, Muse will handle all data conversions entirely automatically. As the Muse steps just use existing (and tested) components, the user does not need to spend time debugging new scripts and learning how to write data conversion applications for the different tools involved.

The clear advantage in this example is not just that the Muse version would take significantly less time to do (as all

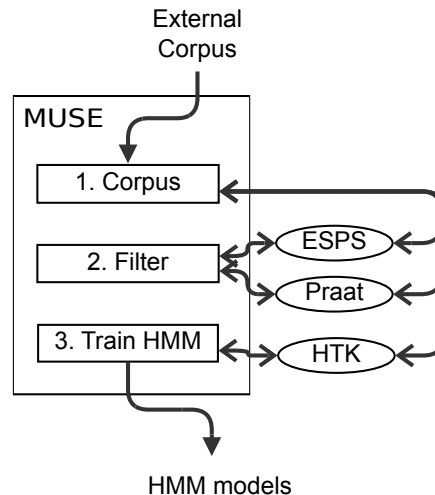


Fig. 2. Training acoustic models from filter parameters in Muse.

required components exist), but that the process is repeatable and flexible. If the user wishes to use an alternative corpus, which may come in a different format and layout, there is no major changes required, step 1 can be used to import corpora from most popular formats. Similarly, if the user decides to control different numbers of states or mixtures in the HMMs, or to use Sphinx instead of HTK, only the options for step 3 need be changed. An additional advantage is that if the corpus is large, a significant amount of time can be saved by the training being performed on a computational cluster, which can be done entirely automatically from Muse.

In situations where not all necessary components exist, it can often be quicker to add the missing component than to implement everything externally without Muse.

6. DISCUSSION AND FUTURE WORK

The Muse platform implements all of the features described in this paper. While the current version of Muse does not contain the necessary components for several sub-domains of speech technology research, the current set of components are useful in a variety of tasks. The release of the platform as open source enables members of the speech community to add components that are useful to them and to contribute to the overall design and development of the platform.

As with any software that wraps around some external applications, situations can arise where the user requires access to all options and the full flexibility of external tools. In some cases, all options may not be available from the current version of Muse. In such situations, it is straightforward (as all Muse tools are scripts and easily editable in any text editor), to add support for the required options. In other cases it can be beneficial to allow the user to interact directly with the underlying toolkit. The HMM-based acoustic model training rou-

tines support such situations, as after models are trained, the generated configurations and data from the training process are stored in a common layout so the user could externally re-train the models with more specific options if required, where the trained models will still function from within Muse without any issues.

Another common issue is that when alternative toolkits are implemented as such, they can function somewhat differently. This is apparent in the *filter:muse* script, where it can combine the results of several filters (which may involve several external tools). Some filters may not support the same frame duration or audio frequency as other filters. In such situations, some filters may not be available due to the specifications of the audio being filtered. As standard, all filters work with 10ms frames and 16kHz speech, but other configurations are significantly dependent on external tools.

The next features to be added include:

1. Speech recognition scripts - Currently scripts are provided to train acoustic models, and use them for force alignment. Scripts will be added to perform basic speech (phone/word) recognition using the trained models.
2. Grapheme to phoneme scripts - These can be used to estimate the pronunciation of any word in a language. The scripts are currently in their final stages of development, where they can automatically learn the pronunciation rules of any language when supplied with a pronunciation dictionary. These are a very useful tool along with the force alignment script to annotate orthographically transcribed corpora.
3. Weighted Finite State Tools - A weighted finite state toolkit, including several training algorithms has already been developed. A front end to manually edit new or trained finite state machines is being developed.
4. Improved Windows support - While the current version is functional under Windows, the system runs best on Mac OS X, Linux and BSD. Some time needs to be spent on testing with Windows Vista and 7 as well.

7. CONCLUSION

This paper introduces the Muse platform for speech technology research. The Muse platform assists the integration of existing speech tools and algorithms. This paper coincides with the release of the platform (available from: <http://muster.ucd.ie/muse>) as open source under the Apache 2.0 license which enables it to be used for both academic and commercial purposes.

The key architecture decisions of the platform are discussed. The components and current features of the platform are presented, where in particular, it helps abstract spoken

corpora, file formats, algorithms, and toolkits. The underlying technology in the platform enable experiments to use multiple processes or be deployed on computational clusters to reduce computation time. New algorithms and support for file formats automatically propagate through the platform, so that any new features become available to all appropriate tools and experiments automatically.

For future work, several new tools will be added, such as: speech recognition scripts, grapheme to phoneme scripts, weighted finite state tools and improved support for the Windows platform. As the platform is available to the public, there may also be several other contributions from other speech technology researchers.

8. ACKNOWLEDGEMENTS

This material is based upon works supported by the Science Foundation Ireland under Grant No. 07/CE/I1142 as part of the Centre for Next Generation Localisation (www.cngl.ie) at University College Dublin. The opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Science Foundation Ireland. Contributors to the Muse platform to date include: Julie Mauclair, Kalu Ogbureke and Robert Kelly.

9. REFERENCES

- [1] S. Young, J. Odell, D. Ollason, V. Valtchev, and P. Woodland, *The HTK book*, Entropic Research Laboratory, Cambridge, England, 1997.
- [2] K.F. Lee, *Automatic speech recognition: the development of the SPHINX system*, Kluwer Academic Pub, 1989.
- [3] Novell, "The Mono Project," <http://www.mono-project.com>, 2010.
- [4] K. Sjölander and J. Beskow, "Wavesurfer-an open source speech tool," in *Sixth International Conference on Spoken Language Processing*. ISCA, 2000.
- [5] E. Klabbbers, K. Stober, R. Veldhuis, P. Wagner, and S. Breuer, "Speech synthesis development made easy: The Bonn Open Synthesis System," *Proceedings of Eurospeech 2001*, pp. 521–524, 2001.
- [6] D. Talkin, *Entropic Signal Processing System (ESPS)*, Entropic Research Laboratory, Cambridge, England, 1993.
- [7] P. Boersma, "Praat, a system for doing phonetics by computer," in *Glott International*, 2002, vol. 5, pp. 341 – 345.
- [8] M. Hall, E. Frank, G. Holmes, Pfahringer, B. Peter Reutemann, and I. Witten, "The weka data mining software: An update," in *SIGKDD Explorations*, 2009, vol. 11.