MODELING SPEECH ANALYSIS WORKFLOWS IN VISTRAILS

Alec Scheuffele

Bachelor of Engineering
Software Engineering

Department of Engineering
Macquarie University

September 3, 2017

Supervisor: Associate Professor Steve Cassidy

ACKNOWLEDGMENTS

I would like to acknowledge my academic supervisor, Associate Professor Steve Cassidy, for his guidance on this thesis project. I also want to thank my parents for their support and encouragement throughout my studies at Macquarie University.

STATEMENT OF CANDIDATE

I, Alec Scheuffele, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Electronic Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment an any academic institution.

Student's Name: Alec Scheuffele

Student's Signature: Alec Scheuffele (electronic)

Date: September 4, 2017

ABSTRACT

Reproducibility of results is a fundamental component of conducting comprehensive research. When experimenting with data-analysis pipelines, time spent managing low-level processes and juggling inconsistencies between tools is a barrier to effective research. Rather than exclude persons who might not have a strong technological/programming background to work-around these limits, utilizing a "higher-level" workflow environment could be of great benefit in certain domains — for this project, that domain is acoustic phonetics. This project is an exploration into the effective use of the workflow software "VisTrails" for building acoustic phonetics workflows. VisTrails is a python-based application where workflows can be visually constructed using a "drag-and-drop" interface of logic modules and linked input/output ports. Users can test workflows under a variety of operational parameters without the need to modify low-level source-code. Furthermore, users can create custom VisTrails packages tailored for particular domains. These can be bundled and shared between colleagues for easily repeatable and extensible experiments. In this project, the core design elements of workflows are investigated within the context of acoustic phonetics, and a target analysis task (classifying phones using formant trajectory data) is implemented at varying levels of module granularity. The benefits/drawbacks of each approach are investigated, and assessments are made on the practicality of working with VisTrails workflows, and their component modules, at these levels of abstraction.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Goals

Acoustic phonetics analysis is the study of the quantifiable acoustic features of human speech. Research in this field often involves passing large audio data-sets through purpose-built computational pipelines, and analyzing outputs. Its a development process that could heavily benefit from some degree of abstraction. Current tools/techniques make reproducing past research unnecessarily difficult. Time spent managing low-level processes, and navigating inconsistencies between tools is a barrier to effective research. Workflow engines are a promising paradigm for addressing many of these issues. Investigating these systems for building and acoustic phonetics analysis workflows would provide some useful insight into the viability of performing such analysis tasks at this level abstraction, and doing so in the most effective way.

VisTrails is a powerful python-based workflow system that facilitates the construction of complex data-analysis pipelines without needing to make constant changes to low-level source code. By linking together networks of draggable input/output modules within a visual workspace, users can quickly investigate different workflows, tweak operation parameters, render interactive output visualizations, and maintain a navigable tree of provenance data for their projects. This project is an exploration into the effective use of the software VisTrails for data analysis in the field of acoustic phonetics.

The goal for this project is to create some sample tools/packages for acoustic phonetics research within VisTrails (located in project repository [25]), construct speech analysis workflows using this package, and assess the benefits/drawbacks of various workflow design decisions. In short, how can VisTrails be most effectively utilized for acoustic phonetics research (steering clear of low-level scripting), what are some of the key workflow component design decisions, and how do these variables impact the efficiency/effectiveness of data analysis tasks.

Key Goals:

- Investigate the effectiveness of VisTrails for building and executing "acoustic phonetics analysis" workflows. (Targeting users who have limited scripting knowledge).

- Create some exploratory acoustic phonetics package modules, build some example workflows using the package, and execute these workflows over test data sets.
- Observe the benefits and drawbacks of various workflow design decisions (e.g. granularity, data-structures) and how they effect the effectiveness of resulting workflows (maintainability, speed, readability, extensibility...etc) within the context of the VisTrails environment.

1.2 Planning

```
Week:  0       1       2       3       4       5       6       7       8       9       10       11       12

        Prep             Development                                                    Analysis
Phase: |---------------|---------------------------------------------------------------|----------------|

       |------0.1------|
               |--0.2--|
                       |-----------1.1---------|
                                               |------------------1.2------------------|
                                               |------------------1.3------------------|
                                                                                       |------2.2-------|
0 - Prep:
    0.1 Review of Knowledge Areas
    0.2 VisTrails Feature Exploration
1 - Development:
    1.1 Developing Core Functionalities, and Integrating Libraries
    1.2 Design and build VisTrails Modules
    1.3 Design and Implement Sample Workflows
2 - Analysis:
    2.2 Analysis and Report
```

1.2.1 Development Strategy

This project was carried out in three sequential phases — Preparation, Development, and Analysis. The majority partition is the development phase, where Acoustic Phonetics workflows are implemented in VisTrails, and the various related modules/sub-workflows are designed and tested. An agile approach was taken for this project phase, employing a flexible interpretation of feature driven development as the development strategy. Features were iteratively implemented and improved upon throughout the development phase as more information was gathered.

The chart above shows a high-level representation of the project schedule. The realization of this schedule was quite flexible, and with some level of overlap. The "Development" phase did not adhere strictly to any kind of development cycle, as iterative features/functionalities were done in a more experimental fashion.

1.2.2 Cost

No costs were incurred with this project, as planned. Tools that were planned/used in completing this project include:

- VisTrails	(free)
- Jupyter Notebook	(free)
- Text Editor (e.g. Notepad, Sublime Text)	(free options)
- Source Control Software (BitBucket)	(free)
- Development Computer	(already own)

1.3 Document Overview

A brief overview of the research tasks performed in this project are described below. These also represent the major sections/topics covered in this paper.

A First Look at VisTrails:
This is an early investigation into the various intricacies of VisTrails. Here, some of the following questions are answered:

- What is a VisTrails module? How do they work?
- What default modules come shipped with VisTrails? What is missing?
- How is control flow handled?
- How are custom VisTrails modules/packages made?
- What are the key elements/variables for designing workflows in VisTrails (e.g. data structures, granularity)?
- What additional (relevant) features/functionalities are included?

Drafting a Target Workflow in Jupyter Notebook:
First, a target acoustic phonetics workflow is chosen (building a classifier for recognizing phones), and this analysis task is implemented in a Jupyter notebook. This scripted workflow serves two main purposes:

- To serve as a testing-ground for integrating the various python speech-analysis libraries/packages into the workflow
- To better understand and debug some of the sub-workflow analysis tasks, having direct scripting access to the data

This section provides a basic overview of the analysis task, and also includes an outline of the Jupyter Notebook implementation.

High Granularity Workflow Implementation in VisTrails:
This is the first attempt at building a speech analysis workflow in the VisTrails environment. The goal was to construct the workflow in its entirety at the same abstraction level as the default data and control-flow modules/packages that come shipped with VisTrails. As a result, this workflow is intended to operate at the highest practical granularity that VisTrails allows. Customized modules are built to integrate the speech analysis libraries, and data libraries. Observations are made with regard to how effective this level of granularity is, how useful the data structures are, and what are (if any) the overall benefits of this workflow implementation (e.g. usability, maintainability, readability, speed, ...etc). From these observations, some insights can be made for the next implementation.

Low Granularity Workflow Implementation in VisTrails:
A workflow is constructed at a much lower granularity using custom speech analysis modules designed to perform the various sub-workflow functionalities from the previous workflow. The same analysis task is performed, but with this new granularity comes certain benefits and pitfalls that merit discussion. These differences are observed and documented.

Analysis and Conclusion:
This section reviews the tasks completed in this project, and draws specific conclusions from the work accomplished. This includes answering questions like:

- What are the major factors that effect the practicality and efficiency of acoustic phonetics workflows?
- What are the major benefits and drawbacks that arise from variances in these factors (readability, maintainability, extensibility, efficiency...etc)?
- What might be the more "ideal" design elements for VisTrails modules/workflows within the context of acoustic phonetics?
- What more could be done to further this research, and how these observations could be applied?

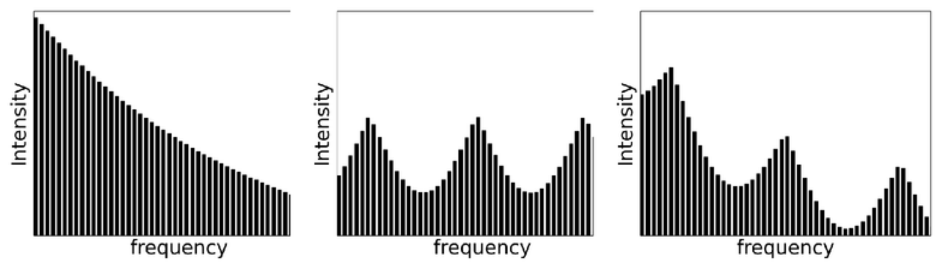
Chapter 2

Background

2.1 Review of Relevent Literature

2.1.1 Acoustic Theory of Speech

The **acoustic theory of speech** explains how humans can create the extensive variation of phonetic sounds that make up vocal languages. A key abstraction is the 'source-filter' decomposition of speech. This breaks speech sounds into two key components — a source sound (vocal chord vibration), and a filter (resonants in the vocal tract). A speaker actively adjusts this resonance filter by changing the shape of their mouth/throat and, consequently, can create different phonetic sounds. [1]



(a) source (b) filter (c) resulting spectrum

Figure 2.1: Source-Filter

While the source vibrations may vary greatly, these differences are overshadowed by the filter function (e.g. whispering vs speaking — phonetic information is retained). Source variations will be noticeable, but have limited effect on the phonetic attributes of the sound. This source-filter principal provides a reference point from which vocal attributes can be measured, quantified, and analyzed.

Mapping quantified phonetic attributes for research cases can provide unique insight into various aspects of human vocalization. These insights can be used in many real-world applications like:

- speaker identification [2]
- speech synthesis
- accurate audio transcription
- speech therapy
- profiling regional dialects [3]
- measuring speech intelligibility [4]

2.1.2 Current Tools/Methods for Acoustic Phonetics Research, and Shortcomings

Current methodologies commonly involve starting from a research question, collecting (or producing) audio samples, extracting relevant information from these samples, and generating output analysis/visualizations. There are many existing tools that perform signal analysis, and map phonetic characteristics. [5] One of the more popular modern tools, ‘Praat’, allows users to write scripts for data-analysis computations/visualizations utilizing the native functionalities of the application. [6] This approach is similar throughout most audio analysis tools, to varying degrees. But these tools have shortcomings with respect to reproducibility, approachability, and extensibility that could possibly be addressed using a different paradigm.

Native scripts are not ideal for sharing ideas — script execution is not intuitive to readers with no prior exposure to the language, or the application running the script. Some less extensible programs may not accomplish certain tasks, may support limited file formats, require running specific releases on older machines, knowledge of deprecated scripts....etc. Time spent managing the lower-level inconsistencies between tools is is a barrier to effective research, especially to those who lack the necessary technical knowledge. [7]

2.1.3 Visual Programming and Workflow Systems

Visual programming offers some promising solutions to the complications mentioned in the previous section. Visual programming systems can be used in a vast number of applications — from gaming, to science, to application development. Unreal Engine (a mainstream game-engine), for example, uses its own visual programming system for defining object oriented classes, allowing users to build everything from surface features/textures

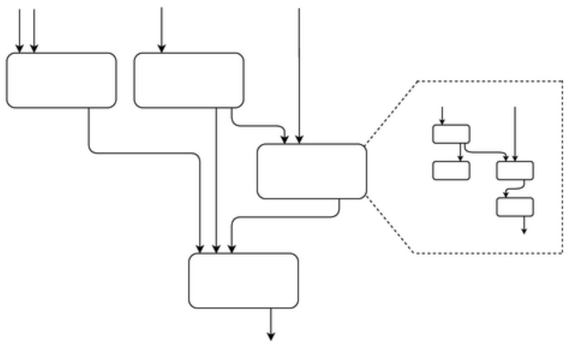


Figure 2.2: Workflow Model

to choreographing in-game events. "GameSalad" [16] is an entirely visual-programming based game-engine where users program 2D games using a drag-and-drop rule-based system. In this paradigm, program execution isn't specified "chronologically", but instead users assign rules to various "actors", and can tweak the parameters of these rules to achieve a desired behavior. It has shown a lot of promise in lowering the barrier-to-entry for building and interacting with complex programs by non-technical users [17]. This approach to programming works well for building games, because many of core processes (e.g. movement, physics, sound, lighting...etc) are common components across a variety of games with only small variations in key parameters. Similarly, many areas of research also utilize a collection of commonly used analysis tasks, suggesting that applying this method of programming to these domains could potentially be very beneficial.

Of particular interest for data-analysis are workflow systems, which can allow users to construct their own tools for building and executing analysis pipelines. Workflows are a sequence of tasks, that when executed from start-to-finish, will accomplish some desired functionality. In visual programming, Workflows can be represented as a series of "modules" linked together through an assortment of input and output ports within a visual workspace. Under the hood these modules may be executing complex processes, but users only need to be concerned with providing an input, handling output, and tweaking specific execution parameters.

Workflow systems assist in interfacing different technologies, joining/extending existing workflows, sharing workflows, outsourcing computationally intensive jobs to external clusters, connecting with various data repositories...etc. Investigations into the effectiveness of particular workflow systems in selected fields is an interesting research question. There are many data-analysis workflow engines that currently exist. These all operate using a similar paradigm, but have different approaches/specializations that might lend themselves better to various fields of research, or to different tiers of users. Some of the more popular systems being developed/explored include:

- 'Taverna' — A 'domain independent' workflow engine in active development (part of the Apache Incubator Project [8]. It has shown great promise for improving portability and reproducibility of research [9], and has already been utilized in many data-intensive research applications. [10]

- Galaxy — The Galaxy platform is a web-based workflow engine [11]. Alveo has adopted galaxy to explore implementing a selection of workflow modules for Speech Analysis. This tool has shown it is capable of building/executing acoustic phonetics workflows, and producing conclusive results. [12] This could provide some initial insight into what functionalities/operations might be useful, and how effective workflow platforms could be in this research area.

- VisTrails — A powerful, open-source, python-based workflow engine. Users can quickly investigate different workflows, tweak operation parameters, render interactive output visualizations, outsource computationally intensive jobs/processes to external servers, and maintain a navigable tree of provenance data for their projects. Everything is built from the ground-up in python, bringing with it library extensibility, and cross-compatibility. VisTrails packages can be developed quickly in pure python, and imported into the platform to be used as workflow modules.

- Jupyter Notebook [24] — A python-script development environment where workflows can be created by building and executing a timeline of modular python scripts and caching the output environment at each step. Jupyter Notebook also includes mark-up tools for adding annotations and output visualizations at each step that can be exported and shared it many readable formats. Jupyter Notebooks are not visually programmed, but are instead scripted within sequential "cells" which are executed in order.

2.1.4 Exploring VisTrails for Acoustic Phonetics Research

VisTrails is very promising candidate for exploring the advantages that such workflow systems might provide to specific areas of scientific research — especially in terms of research reproducibility, and accessibility to users. Analysis tasks often involve passing large sets of sample data (and related semantic information) through standard signal processing algorithms, and analyzing the outputs. Such experimentation might be done more effectively, and in a more approachable manner, using a visually programmed workflow environment where a user doesn't need to learn how to manage low-level source code themselves. If a toolset of interoperable analysis tasks within a given domain could easily be created, exported, and shared, it would offer many advantages in these facets over traditional methods.

Developing and testing an acoustic phonetics package for the VisTrails platform is a promising avenue for advancing tools in this field, improving research reproducibility, and creating a more approachable environment for non-technical users. [13] Apart from the

"Galaxy" project, there have been limited inquiries into the viability of modern workflow platforms specifically within the context of acoustic phonetics research. A key question that surfaced in the existing work is "what is the ideal granularity" of a workflow, and how many modules should it take to perform any particular analysis task. This project will be addressing that gap.

Chapter 3

VisTrails

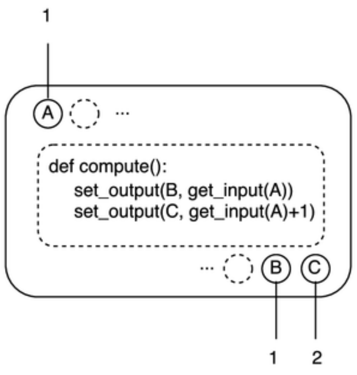
3.1 Introduction

This section is a investigation into the fundemental compenents of VisTrails, how workflows are implemented in VisTrails, and some key design elements to consider when building workflows and workflow modules.

3.2 Workflows in VisTrails

The basic unit of a VisTrails workflow is the VisTrails ' module'. This is the lowest logical element within the context of a Workflow environment. A module consists of 3 basic components: input-ports, output-ports, and a compute function. When a module is executed, it takes the data from the input ports, processes this data in the compute function, and assigns resulting values to the output ports.

Building workflows in VisTrails is a process of chaining together the various input and output modules, which represent the various sequential operations in a computation pipeline. There is no global "state" information that is modified. Instead, all data is passed through the various input/output ports as it makes its way down the pipeline.



```
def compute():
    set_output(B, get_input(A))
    set_output(C, get_input(A)+1)
```

A helpful way of thinking about the execution of a workflow is to trace through starting from the last element. Figure 3.1 depicts this process. The last module will print the value of the element in its input port. That value comes from the output port of the previous module, which is double (x2) the value of the element in *it's* input port. Once the constant at the top of the workflow is reached, this is the value that is passed back down through the pipeline as the functions are executed.

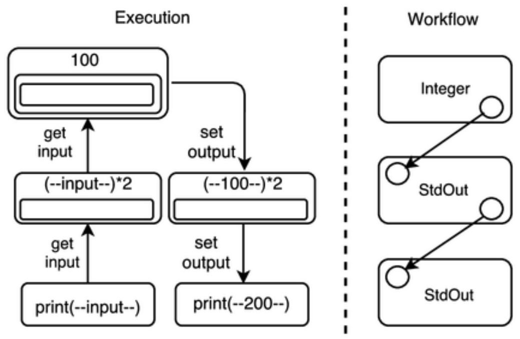
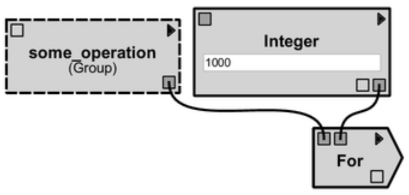


Figure 3.1: Execution

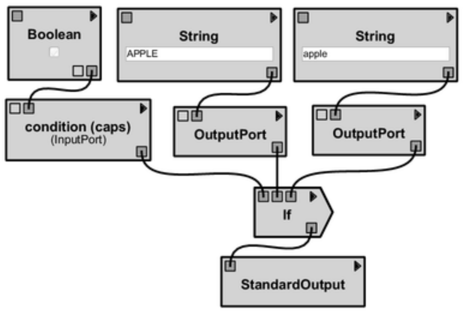
The diagrams in Figure 3.2 show some basic control-flow sub-workflows built in VisTrails. These are built using VisTrails "basic modules" that come packaged with the program. Sub-workflow 3.2b and sub-workflow 3.2c also contain "group" modules. These VisTrails modules represent sub-workflows that have been encapsulated within a single logical container. The input-ports and output-ports of this group will simply be the ports at the top, and the base of the encapsulated pipeline. Using such modules would be required less-often, or perhapse not at all when using a highly abstracted VisTrails a package.

When constructing a workflow at a very low logical level, the creation of looped/iterating subworkflows will be a common necessity. In VisTrails, this consists of creating a subworkflow grouping, and utilizing a VisTrails control-flow module to handle passing and collecting input/output data for this grouping. Using the subworkflow in Figure 3.2c as an example, the diagram below shows what parameters are supplied to the Map module to return "some_function" applied to all values in a list. In the context of VisTrails, some_function is a subworkflow containing one input port (inValue), and one output port (outValue).

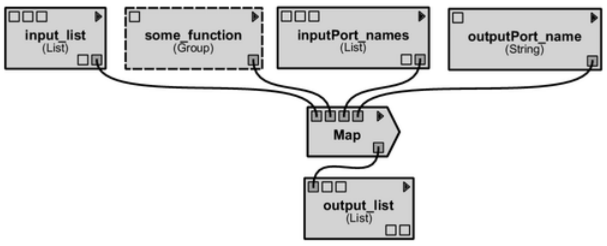
```
          input_port (ID: inValue)  ------> [ pipeline ] ------->  output_port (ID: outValue)

                            MAP parameters:
                            ----------------------------------------------------
                            inputPort =['inValue']
                            outputPort= 'outValue'
                            inputList = [val_1, val_2, va_l3...val_n]
                            functionPort = (link to 'some_function')
                            ----------------------------------------------------

    result = [some_function(val_1), some_function(val_2), some_function(val_3)....some_functionn(val_n)]
```



(a) For Loops



(b) If Statements



(c) Function Mapping

Figure 3.2: Control Flow

Building a complex workflow entirely from the basic-modules would likely be a very messy business. As shown in Figure 3.2, even simple logical structures can require a fairly large number of modules and connectors. Working with complex workflows at such a granularity would be a nightmare to construct, maintain, read, debug...etc. For more complex operations, VisTrails allows programmers to create their own custom modules within a "python source" module. The programmer simply needs to specify the input/output ports (and their data types), and they can define their own compute function in plain-python. Using the python-source module, workflows can be created with almost *any* level of granularity.

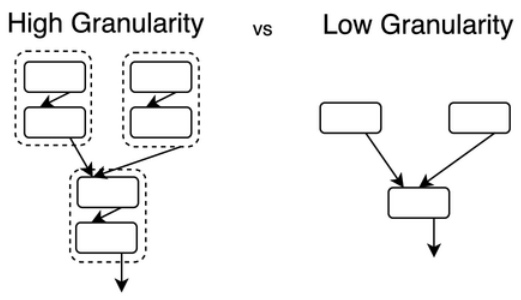
The basic python-source module is meant to be used as a one-time solution component in a workflow. Reusable modules can be scripted outside of VisTrails, and then imported into VisTrails as a package. This process also allows a programmer to define new data types that can be passed between modules, or required by input/output ports. Once imported into VisTrails, the modules within a package can be dragged into the workflow from the "modules" pane of the VisTrails UI. These custom packages/modules can be useful in many scenarios — from simply interfacing between basic modules, to providing complex high-level functionalities for specific applications (e.g. acoustic phonetics analysis).

The ability for a user to integrate *both* packaged modules, and script (if desired) is a really powerful feature of VisTrails. The wasted hours spent implementing complex workarounds to design restrictions in a toolset can be entirely avoided. If a user knows some basic scripting fundamentals, they have the freedom to use plain python at any point in the workflow.

This is possible because VisTrails is implemented entirely in python. All Modules in VisTrails are simply python classes at their core. As a result, the extensibility of VisTrails is enormous. Almost any python package/library could be integrated with VisTrails. Any VisTrails workflow could, technically, be considered a python program—but this program includes a very heavy infrastructure/dependency stack that it always requires for execution (a.k.a. the VisTrails program). This is one of the more pronounced constraints imposed by VisTrails — VisTrails workflows require a copy of VisTrails to run. The exporting and sharing of workflows can be easily accomplished using an XML representation (generated by VisTrails), but a colleague receiving such a file would still need the VisTrails software to import and execute the pipeline.

3.3 Designing Workflow Modules

There are two key workflow characteristics that have the largest impact on the practicality and the effectiveness of a workflow, or any workflow system package: the underlying data structure, and the module granularity.



Granularity — The granularity of a workflow defines the degree to which a workflow, or a toolset, is composed of smaller sub-components. A high-granularity workflow utilizes a denser collection of modules to perform some subworkflow task. Granularity is determined, largely, by the design of modules in a toolset, and emerges as a direct result of how these modules can be used together to perform analysis tasks. There are some obvious trade-offs when it comes to workflow granularity. While it may be very concise to encapsulate a large sequence of logical steps within a single module, this places significant restrictions on the flexibility/utility of modules, as the results of intermediate processes are unaccessible.

Approaching an ideal granularity when designing workflow modules is a tricky task. The less granularity there is, the more restricted the toolset becomes. But with increased granularity, a workflow can become chaotic. In a uniform toolset, as the number of required modules for any subworkflow task increases, this increase is likely mirrored throughout all logical structures of similar complexity Workflows should ideally utilize the *least* number of modules necessary to perform a task, and the modules themselves should offer maximum functionality and interoperability. Finding a balance between maximum design freedom, and conciseness, is a challenge.

Low granularity workflows would, in theory, be well suited for performing more specialized data-analysis tasks under a multitude of different parameter configurations. Modules in these workflows may be designed with a large number of input ports for adjusting these parameters and conducting numerous tests. High granularity is more versatile. However, this versatility in design space comes at a cost. Because modules are less specialized, certain adjustments to workflow execution (done by simply toggling a parameter for low-granularity workflows) may now require an entire rebuild of certain partitions of the workflow. Also, the computational overhead should be considered for each module. While iterative processes executed as scripts might compute very quickly, looping through a large number of modules in VisTrails takes much longer.

Data Structure — A *key* element in the workflow design is the underlying data-structure used to build, organize, and query sample data. Designing modules that allow a user to most effectively interact with this structure (without using source-code) is a challenge.

When a workflow is executed, large collections of data are processed in bulk as they pass through the pipeline. The only interface a module has to the data is through its input and output ports. Sample data likely includes "meta" information that must also pass through each module before finally being utilized at some later stage (for example: speaker, age, gender ...etc). This means that the sample data needs to be stored in a structure that can contain (and retain) all the necessary relational information during workflow execution,

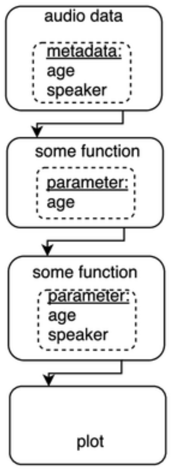


Figure 3.3: Concise and Extensible

and can do so in a way that is easily accessible to a user *and* is maximally compatible will all other modules and python libraries.

A naive approach that does not address the desired characteristics mentioned above would be to manage data in structured lists that are curated specifically for the function of any particular module. For example, an input port may take a list of "phone trajectories" as an input, which are stored like this:

```
[speaker, [phone, [[f1][f2][f3]]]]
```

But this is a very BAD approach. What if your trajectory data only had the first two formants (f1, f2)? what if you needed to include extra tag information for each trajectory (e.g. gender), or less — where would this information go? How could you properly query such data in an sensible and extensible way, for any combination of meta information?

Tables provide a much more extensible may of managing data. Using tables, you can store and query various attributes, perform table joins, interface with external table-based tools, and have a "universal" structure for passing and operating over data. Entire tables can be passed to a function module, and the name of relevant data columns can be passed as parameters. Alternatively (but logically identical), the columns can be selectively pulled from an external table outside of a module, and *then* can be passed to the associated input ports. This concept can be visualized in figure 3.4.

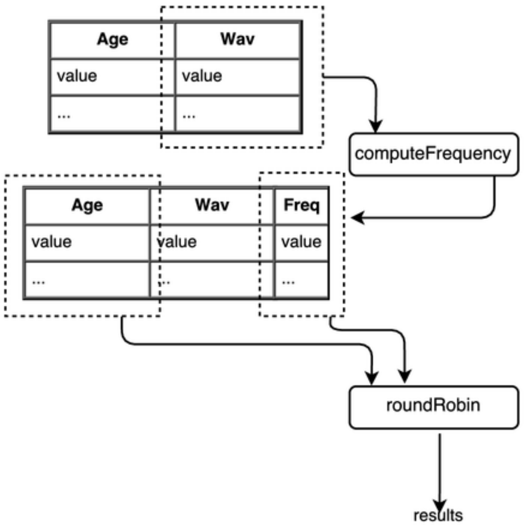


Figure 3.4: Table Data Flow

VisTrails has a native ‘tabledata’ package included that contains modules for performing very basic table operations. However, the functionality provided from these modules is very limited. Making advanced queries, complex joins...etc simply cannot be performed within the workflow using these modules. To do so would require messy workarounds that quickly make the workflow chaotic and difficult to maintain. Its clear that the native table package was not designed to be constantly modified throughout a workflow. Even using “python source”, the functionalities is not comprehensive, and are not concise.

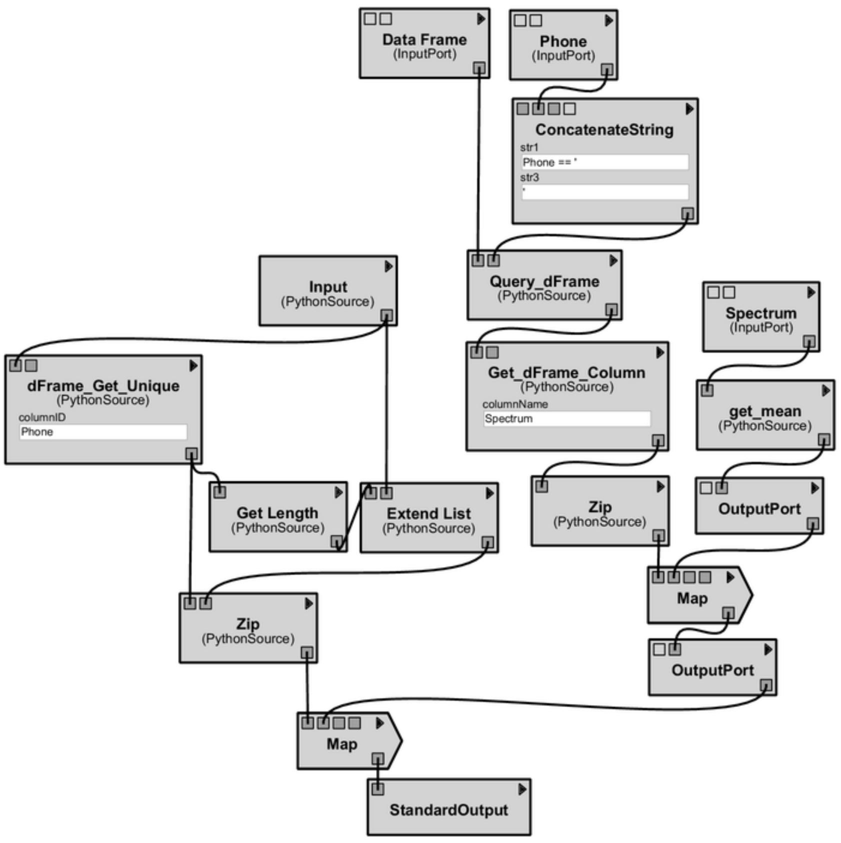
The Pandas [15] library solves some of these problems. Pandas is a data-structures and data-analysis toolset that includes a large suite of operations that can be used to build, interact with, and manipulate table data very efficiently. For this project, Pandas will be used under the hood for managing data within each module. If the VisTrails ‘tabledata’ package ever *did* need to be used (possibly for integrating other native VisTrails elements utilizing the tabledata package), it would be trivial to convert between the Pandas ‘DataFrame’ object, and the VisTrails ‘Table’ object.

However, while it offers a vast range of functionalities in the scripting realm, maintaining the accessible versatility of this package *without* scripting could be a challenge. To an experienced programmer with strong foundation of Pandas knowledge, layered data operations can be specified very concisely in a scripting environment, but implementing sets of modules to perform these same functionalities might not be so concise. Certain tasks in Pandas (such as applying a custom function to a group-by data object) can be done with very few lines of code using the “apply” function. But without the ability to create a passable function for Pandas, an opportunity for conciseness and efficiency is lost. “Subworkflow” processes can be constructed in VisTrails, and can be utilized by control-flow modules (like “map”) by passing related input and output ports as parameters — but the subworkflow itself does not exist as a callable “function”.

To illustrate this point, consider the example below. This subworkflow returns the averaged spectrums of all spectrum data for each phone in a dataset. A user familiar with Pandas can accomplish this with a few lines of code (in fact, its possible with just a single line of code). However, a high granularity workflow implementation of this task shown in Figure 3.5 must take a slightly different approach. Instead, it loops over nested subworkflows that query the Pandas DataFrame to group each “Phone”, and then calculate the mean of the paired Spectrum data. This implementation may not as efficient or concise as the scripted version shown in Figure 3.1, but was still possible to accomplish in its entirely without needing to write a single script.

Related figure and script 3.5 and 3.1 can be seen on the following page.

Figure 3.5: Average Spectrum for each Phone



Listing 3.1: Python-Source Implementation

```
def spectrumMean(spectrums):
    return [np.mean(x) for x in zip(*map(list, spectrums))]

Result = result_dFrame.groupby('Phone')['Spectrum'].apply(list).apply(spectrumMean)
```

Chapter 4

Building a Classifier for Recognizing Phones

4.1 Introduction

Before building anything in VisTrails, it is important to make sure there is a solid foundation "underneath the hood". While the end-goal is to build and execute workflows without the need for low-level scripting, the effectiveness of the data structures and the interoperability of the libraries being used are more efficiently tested outside of VisTrails in a more familiar environment. In this case, a target workflow (training and testing a classifier to recognize spoken phones) is chosen as the primary analysis task.

This analysis task, taken from the paper "Dynamic Features in Children's Vowels" [14] was selected because it consists of some fundamental signal analysis and machine learning tasks, and requires effective management of sample data/meta-data throughout the pipeline — both important for assessing the effectiveness of VisTrails within the context of acoustic phonetics.

The scripted workflow will be constructed in a **Jupyter Notebook**. Jupyter Notebook is a python-script development environment where workflows can be created using a timeline of modular scripts, and the output environment is cached at each step. It also includes mark-up tools for adding annotations and output visualizations. This is a useful environment to draft the target workflow in a more traditional scripting approach, and still offers some great features (e.g. caching, annotations, modularization) that can speed up testing, and maintains the characteristics of a data-analysis "workflow".

This section walks through the Jupyter Notebook implementation of the target-workflow, and outlines the major logical processes at each stage. Abbreviated snapshots of the data between each step in the workflow are also provided.

4.2 Overview

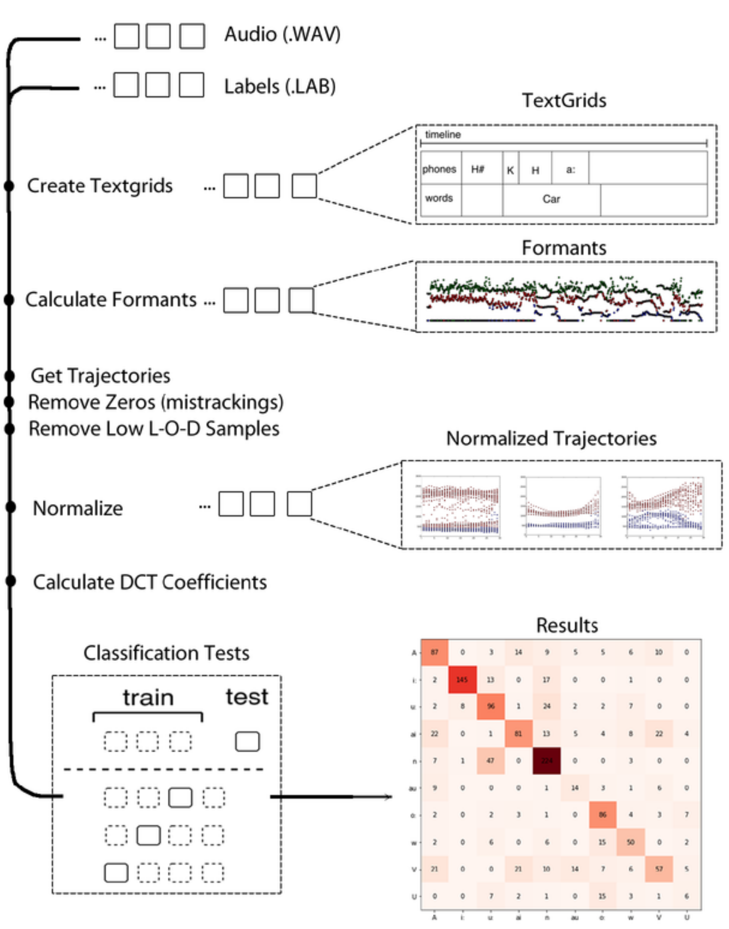
4.2.1 Analysis Task

The analysis task for this workflow is to train and test a classifier that will "classify spoken phones using formant trajectory data". Input data consists of WAV files and associated label files (specifying the interval locations of phonetic attributes). There are numerous WAV/label pairs of different phrases for each speaker, with a total of 8 speakers. The output of this workflow will be a confusion matrix representing the performance of the classifier. The confusion matrix of an effective classifier will have weights concentrated in the cells running diagonally across the matrix (where the row index matches the column index). Cell location represents the "expected value" vs the "output value".

4.2.2 Key Processes

- **Step1: Convert label-file interval data into TextGrid files** — Label files are parsed and the TGT package is used to build a python TextGrid object. These TextGrid objects can be passed around, or saved to the file system. This step is not completely necessary, as the label files already contain all the desired information for this workflow — However, because TextGrids are highly compatible and modifiable thanks to the TGT package, this step has other potential use cases.
- **Step2: Compute the first 3 formants of the WAV files using WRASSP** — The formant timeline for the entirety of each WAV file is calculated, and returned as a formant object.
- **Step3: Pull formant trajectories for each phone** — Using interval information specified in the TextGrid objects, the formant trajectories (formant time-line for a specific interval) are pulled from the formant outputs of the previous step.
- **Step4: Discard zero values, and normalize the trajectory data** — Formants that have been mis-tracked occasionally record false "zero-value" measurements scattered throughout a timeline. These must be removed, and the ranges of all trajectories are normalized, to prepare them for classification.
- **Step5: Conduct round-robin classifier training and testing** — First, the first 3 DCT (discrete cosine transform) coefficients of the first 3 formants in the trajectory data are calculated using SciPy Fast Fourier Transform (FFT) library. These DCT coefficients retain the characteristic features of the trajectory data, but represent them in a compact form that a classifier can more effectively work with. Then, a series of round-based classification tests are conducted where each speaker has a turn being the test-set while the rest of the speaker data is used as the training set.
- **Step6: Analyze results** — Build a confusion matrix from the results of the training/testing rounds, and analyze the accuracy of the classification results.

```
------------------------------INPUT:--------------------------------------------OUTPUT--------------
|                                                             |                                    |
|  speakers:             Sp1               Sp2     ...     sp6 |            X     Y                 |
|                         |                                   |         +----+----+                |
|                         |                                   |       X |  1 |  0 |                |
|  files:     [phrase_1.WAV, phrase_1.LAB],                    |         +----+----+   ...          |
|             [phrase_2.WAV, phrase_2.LAB],                    |       Y |  0 |  1 |                |
|                        ...                                   |         +----+----+                |
|                                                             |            ...    \                |
|                                                             |                 correct            |
-----------------------------------------------------------------------------------------------------
```



	A	i:	u:	ai	n	au	o:	w	V	U
A	87	0	3	14	9	5	5	6	10	0
i:	2	145	13	0	17	0	0	1	0	0
u:	2	8	96	1	24	2	2	7	0	0
ai	22	0	1	81	13	5	4	8	22	4
n	7	1	47	0	224	0	0	3	0	0
au	9	0	0	0	1	14	3	1	6	0
o:	2	0	2	3	1	0	86	4	3	7
w	2	0	6	0	6	0	15	50	0	2
V	21	0	0	21	10	14	7	6	57	5
U	0	0	7	2	1	0	15	3	1	6

Figure 4.1: High-Level Visualization of Analysis Task

4.3 Core functionalities and Libraries

- **Wrassp:** [19] An "R" wrapper around the c-library "libassp" (Advanced Speech Signal Processor). This provides support for complex signal processing operations currently not available in pure-python libraries. (e.g. formant tracking)
- **SciPy:** [23] Useful modules for scientific calculations. Particularly pertinent is SciPy.signal, which contains some signal processing functions (e.g. fft - Fast Fourier Transforms).
- **NumPy:** [22] Python implementations of multi-dimensional array structures, and a suite of high-level operations to perform over these arrays, resulting in execution times more closely resembling that of compiled code.
- **Pandas** Data-structures and data-analysis toolset that includes a large suite of operations that can be used to build, interact with, and manipulate indexed data.
- **Tgt:** [20] Reads and writes Praat TextGrid files. Textgrid files will be used to represent semantic and syntactic information about any audio data being analyzed.
- **SciKitLearn:** [21] Scikit learn is a machine learning library. It will be used to create, train, and test classifiers.

The libraries listed above provide many algorithms and data-structures for managing audio data and attributes, as well as performing standard speech/signal analysis computations (e.g. formant tracking). These will be used interoperability with each other within the VisTrails workflow environment. Various interfaces and helper methods will also need to be implemented to supplement these tools (e.g. normalizing data, pulling phone trajectories, round-robin training/testing...etc)

4.4 Design and Draft in Jupyter Notebook

Setup

```
#----------------------------imports---------------------------------
%matplotlib inline
import tgt as tgt
import matplotlib.pyplot as plt
import os
from scipy import fftpack
from sklearn import svm
from sklearn.naive_bayes import GaussianNB
import subprocess
import pandas as pd
import numpy as np
from IPython.display import display
```

```
#------------------------------setup-----------------------------------
workspacePath = "/Users/Alec/Desktop/children"
speakers = ["sp1","sp2","sp3","sp4","sp5","sp6","sp7","sp8"]
targetPhones = set(['u:', 'A', 'ai', 'o:', 'au', 'V', 'U', 'i:', 'w', 'n'])
```

Step1: Build Textgrids from label files

Here is where the label files will get converted into textgrid files. This is done simply by walking through the tabbed label values and recording the specified intervals. Then, the TGT package is used to build a python textgrid object that can be passed around, or saved to the file system. This step is not completely necessary, as the lab files already contain all the desired information. However, because TextGrids are highly compatible and modifiable thanks to the TGT package, this step has its merits.

```
#----------------function to create textGrid from label file------------
def labToTextgrid(labFile_fileObject, ID=""):
    labFileData = [str(line).splitlines()[0].split("\t") for line in labFile_fileObject.
        ↪ readlines()]
    newTextGrid = tgt.TextGrid(ID)
    phnTier = tgt.IntervalTier(name="PHN")
    startTimeIndex = 0
    for stamp in range(3,len(labFileData)):
        phnTier.add_annotation(tgt.Annotation(float(startTimeIndex), float(labFileData[
            ↪ stamp][1]), str(labFileData[stamp][3])))
        startTimeIndex = labFileData[stamp][1]
    newTextGrid.add_tier(phnTier)
    return(newTextGrid)


#---------------execute creation of textGrids from label files------------
textgrid_dFrame = pd.DataFrame([], columns=['ID', 'Speaker', 'TextGrid'])
for speaker in speakers:
    labFileDirectory = workspacePath+"/"+speaker+"/labels"
    labelFileNames = [n[0]+"."+n[1]+"."+n[2] for n in [f.split(".") for f in os.listdir(
        ↪ labFileDirectory)] if n[-1] == 'lab']
    for labFile in labelFileNames:
        ID = labFile.split('.lab')[0]
        textGrid = labToTextgrid(open(labFileDirectory+"/"+labFile, "r"), ID)
        textgrid_dFrame = textgrid_dFrame.append(pd.DataFrame({'ID':[ID], 'Speaker':[
            ↪ speaker], 'TextGrid':[textGrid]}))

display(textgrid_dFrame[:3])
print textgrid_dFrame[:3].to_latex()
```

	ID	Speaker	TextGrid
0	sp1.1249	sp1	((Annotation(0.0, 1.9075, "H#"), Annotation(1....
0	sp1.1457	sp1	((Annotation(0.0, 1.68002, "H#"), Annotation(1...
0	sp1.2121	sp1	((Annotation(0.0, 2.3801, "H#"), Annotation(2....

Step2: Track formants for each Wav file

In this stage, the first 3 formants are computed for every WAV file in the data set. “Formant Maps” containing the first 3 formants are currently stored in a Formant class shown below. It also includes a method useful for pulling out formant slices between specified intervals.

The getFormantsWRASSP() function utilizes the WRASSP library (through a subprocess call to R) to compute the first three formants and return them as a Formant object.

```
class Formant(object):
    #formantData: -> [-----------------------------------------------------------------]
    #                 [--timestamp1--], [-----f1-----], [-----f2-----], [-----f3-----]
    #                    val1, val2...
    #
    def __init__(self, formantData):
        self.timestamp = formantData[0]
        self.f1 = formantData[1]
        self.f2 = formantData[2]
        self.f3 = formantData[3]

    def pullIntervalsFromTimeline(self, startTime, endTime):
        f1_pulled = []
        f2_pulled = []
        f3_pulled = []
        for i in range(0, len(self.timestamp)):
            if (self.timestamp[i] > startTime and self.timestamp[i] < endTime):
                f1_pulled.append(self.f1[i])
                f2_pulled.append(self.f2[i])
                f3_pulled.append(self.f3[i])
        return [f1_pulled, f2_pulled, f3_pulled]

    def get_timestamp(self):
        return self.timestamp
    def get_f1(self):
        return self.f1
    def get_f2(self):
        return self.f2
    def get_f3(self):
        return self.f3
```

```
#--------------------------------map formants---------------------------------
def getFormantsWRASSP(wavPath, startTime, endTime):
    formantData = []
    scriptDir = '/Users/Alec/Documents/MacUniFiles/Thesis/PlainPython/R_Scripts/formTest.
        ↪ R'
    output = subprocess.check_output(['Rscript',
                                      scriptDir,
                                      wavPath,
                                      str(startTime),
                                      str(endTime)],
                                      shell=False).split('␣', 2)
    numberOfFormantSlices = int(output[0])
    formantSliceDuration = 1/float(output[1])
    dataList = map(int, output[2].split())
    numberOfFormants = 3
    formantData.append([formantSliceDuration*i + formantSliceDuration/2 for i in range(0,
        ↪  numberOfFormantSlices)])
    formantData.extend([dataList[i*numberOfFormantSlices : (i+1)*numberOfFormantSlices]
        ↪ for i in range(0, numberOfFormants+2)])
    return Formant(formantData)


#------------------Execute: Compute Formants for each Wav----------------------
formants_dFrame = pd.DataFrame([], columns=['ID', 'Speaker', 'TextGrid', 'Formant_Map'])
```

```
for index, row in textgrid_dFrame.iterrows():
    textGrid = row['TextGrid']
    wavPath = workspacePath+"/"+row['ID'].split('.')[0]+"/"+"wavs"+"/"+row['ID']+".wav"
    formantMap = getFormantsWRASSP(wavPath, textGrid.start_time, textGrid.end_time)
    formants_dFrame = formants_dFrame.append(pd.DataFrame({'ID':row['ID'], 'Speaker':row[
        ↪ 'Speaker'], 'TextGrid':row['TextGrid'], 'Formant_Map':[formantMap]}))
display(formants_dFrame[:3])
```

	Formant_Map	ID	Speaker	TextGrid
0	__main__.Formant object	sp1.1249	sp1	((Annotation(0.0, 1.9075, "H#"), Annot....
0	__main__.Formant object	sp1.1457	sp1	((Annotation(0.0, 1.68002, "H#"), Annot...
0	__main__.Formant object	sp1.2121	sp1	((Annotation(0.0, 2.3801, "H#"), Annot....

Step3: Get formant trajectories of phones

Using the intervals specified in the TextGrid objects, and formant values calculated in the previous step, pull out the formant trajectories for each target phone present in the data.

```
#---------------------getting phone formant trajectories--------------------------
def getPhoneIntervals_fromTextgrid(textgrid, target, phoneTier):
    vowelTier = textgrid.get_tier_by_name('PHN')
    vowelAnnotations = vowelTier.get_annotations_with_text(pattern=target, n=0, regex=
        ↪ False)
    intervals = [[float(annotation._start_time), float(annotation._end_time)] for
        ↪ annotation in vowelAnnotations]
    return intervals


def getPhoneTrajectory_fromTextgrid(textgrid, formantMap, target, phoneTier):
    return [formantMap.pullIntervalsFromTimeline(interval[0], interval[1])
            for interval in getPhoneIntervals_fromTextgrid(textgrid, target, phoneTier)]



#----------------Execute: Compute trajectories for each phone---------------------
trajectories_dFrame = pd.DataFrame([], columns=['ID', 'Speaker', 'Phone', 'f1_traj', '
    ↪ f2_traj', 'f3_traj'])
for index, row in formants_dFrame.iterrows():
    textGrid = row['TextGrid']
    targetsPresent = list(set([str(interval.text) for interval in textGrid.
        ↪ get_tier_by_name('PHN')]) & targetPhones)
    wavPath = workspacePath+"/"+row['ID'].split('.')[0]+"/"+"wavs"+"/"+row['ID']+".wav"
    formantMap = row['Formant_Map']
    for target in targetsPresent:
        trajectories = map(list, zip(*getPhoneTrajectory_fromTextgrid(textGrid,
            ↪ formantMap, target, 'PHN')))
        ID_list = [row['ID'] for trajectory in trajectories[0]]
        phone_list = [target for trajectory in trajectories[0]]
        speaker_list = [row['Speaker'] for trajectory in trajectories[0]]
        trajectories_dFrame = trajectories_dFrame.append(pd.DataFrame({'ID':ID_list, '
            ↪ Speaker':speaker_list, 'Phone':phone_list, 'f1_traj':trajectories[0], '
            ↪ f2_traj':trajectories[1], 'f3_traj':trajectories[2]}))
display(trajectories_dFrame[:3])
```

	ID	Phone	Speaker	f1_traj	f2_traj	f3_traj
0	sp1.1249	u:	sp1	[271, 327, 336 ...	[0, 0, 2323...	[2513, 2499, 2522...
0	sp1.1249	ai	sp1	[350, 389, 393 ...	[1104, 1081, 1049...	[2139, 2240, 1974...
0	sp1.1249	n	sp1	[361, 354, 348 ...	[1299, 1444, 1354...	[2451, 2454, 2454...

Step4: Discard zero values, and normalize the trajectory data.

This step goes through all of the calculated trajectories and “normalizes” them. This makes all of the trajectory data fit within a fixed, uniform size, while maintaining the geometry of the trajectory. Any zero values that have been mis-tracked are also removed.

The first 3 DCT coefficients are then calculated using the SciPy fft library — these values will be used to train the classifier in the next step. These DCT coefficients allow for the characteristics of the trajectory data to be represented in a more succienct form.

```
#--------------------function to normalize trajectories--------------------------------
def normalizeFormantTimeSeries_Interpolate(formantValues, factor):
    result = []
    offsetRatio = float(len(formantValues)-1)/float(factor-1)
    for i in range(0,factor):
        offsetIndex = i*offsetRatio
        valueRange = formantValues[min(len(formantValues)-1,int(offsetIndex)+1)] -
            ↪ formantValues[int(offsetIndex)]
        offsetAmount = offsetIndex - int(offsetIndex)
        result.append(formantValues[int(offsetIndex)] + (valueRange*offsetAmount))
        #value      =                   ---index---    +     ---offset amount---
    return result



#----------------------normalize data and compute DCT coefficients---------------------
normalizedTraj_dFrame = trajectories_dFrame.filter(['ID', 'Speaker', 'Phone'], axis=1)
normalizedTraj_dFrame['f1_traj_normalized'] = trajectories_dFrame['f1_traj'].apply(lambda
    ↪ x: normalizeFormantTimeSeries_Interpolate([y for y in x if y>0], 40))
normalizedTraj_dFrame['f2_traj_normalized'] = trajectories_dFrame['f2_traj'].apply(lambda
    ↪ x: normalizeFormantTimeSeries_Interpolate([y for y in x if y>0], 40))
normalizedTraj_dFrame['f3_traj_normalized'] = trajectories_dFrame['f3_traj'].apply(lambda
    ↪ x: normalizeFormantTimeSeries_Interpolate([y for y in x if y>0], 40))
normalizedTraj_dFrame['dct'] =
        normalizedTraj_dFrame['f1_traj_normalized'].apply(lambda x: fftpack.dct(x)[0:3].
            ↪ tolist()) + \
        normalizedTraj_dFrame['f2_traj_normalized'].apply(lambda x: fftpack.dct(x)[0:3].
            ↪ tolist()) + \
        normalizedTraj_dFrame['f3_traj_normalized'].apply(lambda x: fftpack.dct(x)[0:3].
            ↪ tolist())

display(normalizedTraj_dFrame[:3])
print normalizedTraj_dFrame[:3].to_latex()
```

	ID	Speaker	Phone	f1_traj_normalized	f2_traj_normalized	f3_traj_normalized	dct
0	sp1.1249	sp1	u:	[271.0, 335.53846,..	[2323.0, 2202.56410256..	[2513.0, 2520.8205..	[37391.7435897, 438.69..
0	sp1.1249	sp1	ai	[350.0, 392.38461,..	[1104.0, 1050.64102564..	[2139.0, 1987.6410..	[32995.5384615, -1846...
0	sp1.1249	sp1	n	[361.0, 355.61538,..	[1299.0, 1410.53846154..	[2451.0, 2453.3076..	[27348.0, 624.94403786..

Step5: Train Classifier

In this step, “round robin” training and testing is performed on data set, where each speaker has a turn being the test-set while the rest of the speaker data is used as the training set. A classifier constructor is passed as a parameter to the round-robin function. With this, a classifier of any type is created for each speaker training/test group and stored in “classifiers”. These classifiers are then trained for their respective roles in the round-robin testing. This is illustrated below:

```
classifiers: [classifier1, classifier2, classifier2, ...]
ClassificationData = [speaker]
                            |
                            [inputs, outputs]
                               |        |
                               |    expected phone
                               |
                               [dct(f1_traj) + dct(f2_traj), dct(f3_traj)]

----------------------------------------------------------------------------
round1:
classifier:  classifier1
train:       [ClassificationData[1], ... , ClassificationData[numOfSpeakers]]
test:        ClassificationData[0]
----------------------------------------------------------------------------
round2:
classifier:  classifier2
train:       [ClassificationData[0], ... , ClassificationData[numOfSpeakers]]
test:        ClassificationData[1]
----------------------------------------------------------------------------
...
```

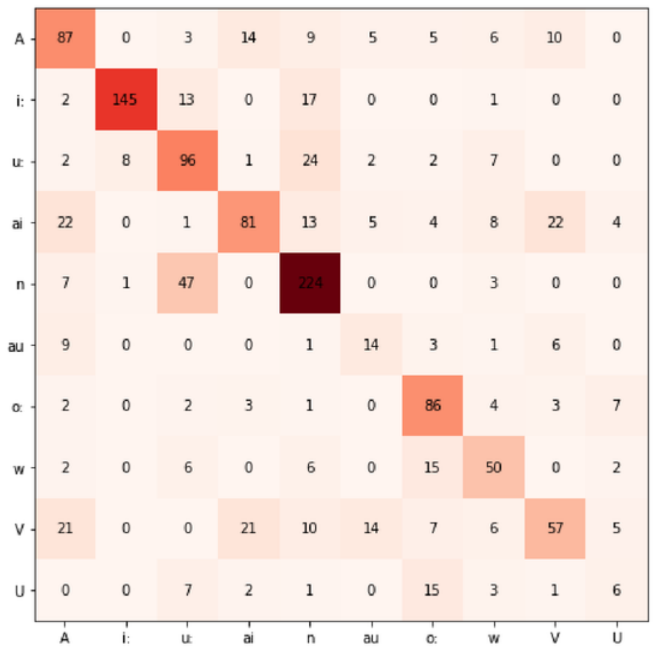
```
#-------------------function to train classifier (round robbin)--------------------
def roundRobbin(dataFrame, factorID, dataIn, dataOut, classifierConstructor):
    factorValues = dataFrame[factorID].unique().tolist()
    classifiers = [classifierConstructor for factor in factorValues]
    results = []
    for i in range(0,len(factorValues)):
        trainingData = [list(dataFrame[dataFrame[factorID] != factorValues[i]][dataIn]),
            ↪ list(dataFrame[dataFrame[factorID] != factorValues[i]][dataOut])]
        testData = [list(dataFrame[dataFrame[factorID] == factorValues[i]][dataIn]), list
            ↪ (dataFrame[dataFrame[factorID] == factorValues[i]][dataOut])]
        classifiers[i].fit(trainingData[0], trainingData[1])
        predictions = classifiers[i].predict(testData[0])
        results.append([predictions, testData[1]])
    return results
```

```
#------------------------------train over data---------------------------------
predictions_bySpeaker = roundRobbin(normalizedTraj_dFrame, 'Speaker', 'dct', 'Phone',
    ↪ GaussianNB())
allInputs = [item for sublist in [x[0] for x in predictions_bySpeaker] for item in
    ↪ sublist]
allOutputs = [item for sublist in [x[1] for x in predictions_bySpeaker] for item in
    ↪ sublist]
confusionMatrix = buildConfusionMatrix(allInputs, allOutputs)
```

Step6: Analyze the classification results

Using a confusion matrix, the effectiveness of the classifier can be easily visualized, and quantified. Confusion matrices, in particular, allow for possible patterns/relationships in mis-classifications to be seen. The code below generates a confusion matrix (as a simple 2d array), and displays the results both numerically, and as a heat-map.

```
#-----------------------function to Generate a Confusion Matrix----------------------
def buildConfusionMatrix(result, expected):
    predictionMatrixIndexing = {}
    confusionMatrix = [[0 for x in range(0, len(targetPhones))] for x in range(0, len(
        ↪ targetPhones))]
    targetPhonesList = list(targetPhones)
    for index in range(0, len(targetPhonesList)):
        predictionMatrixIndexing[targetPhonesList[index]] = index
    for i in range(0,len(result)):
        confusionMatrix[predictionMatrixIndexing[expected[i]]][predictionMatrixIndexing[
            ↪ result[i]]] += 1
    return [targetPhones,confusionMatrix]



#-------------------generate confusion matrix, and render heat map------------------

plt.figure(figsize=(8, 8))
column_labels = list(confusionMatrix[0])
row_labels = list(confusionMatrix[0])
plt.xticks(range(len(row_labels)), row_labels, size='medium')
plt.yticks(range(len(row_labels)), row_labels, size='medium')


#annotations
for row in range(0, len(confusionMatrix[1])):
    for col in range(0, len(row_labels)):
        plt.text(col , row, confusionMatrix[1][row][col],  horizontalalignment='center',
            ↪ verticalalignment='center', size='medium')
plt.imshow(confusionMatrix[1], cmap='Reds', interpolation='nearest')
plt.show()
```



	A	i:	u:	ai	n	au	ɑ:	w	V	U
A	87	0	3	14	9	5	5	6	10	0
i:	2	145	13	0	17	0	0	1	0	0
u:	2	8	96	1	24	2	2	7	0	0
ai	22	0	1	81	13	5	4	8	22	4
n	7	1	47	0	224	0	0	3	0	0
au	9	0	0	0	1	14	3	1	6	0
ɑ:	2	0	2	3	1	0	86	4	3	7
w	2	0	6	0	6	0	15	50	0	2
V	21	0	0	21	10	14	7	6	57	5
U	0	0	7	2	1	0	15	3	1	6

Figure 4.2: Output Confusion Matrix

Looking at the confusion matrix generated above, we can see that the classifier is accurately classifying phones, and the workflow produces the expected output. At this stage, the desired workflow has successfully been implemented in python-source, and the the libraries/packages are functioning as desired. Now, the next step will be to recreate this as a VisTrails workflow. The idea is to achieve identical functionality, but by using only VisTrails modules, and not needing to write any python scripts. This task is covered in the next section.

Chapter 5

VisTrails Implementations

5.1 Introduction

In the previous chapter, a target workflow was successfully built in Jupyter Notebook. Now, the task is to translate this python-source pipeline into a VisTrails workflow. There will be two major implementations — one will work at a high-granularity, and the other will be at a low-granularity. By analyzing the strengths/drawbacks of each workflow, this exercise can provide insight into what an "ideal" granularity might be for acoustic phonetics workflows, and also how the tools that VisTrails provides can be more effectively utilized within these constraints. XML encodings of the final workflows, and the python-package of their component modules can be found in the project repository [25].

5.2 High Granularity Workflow

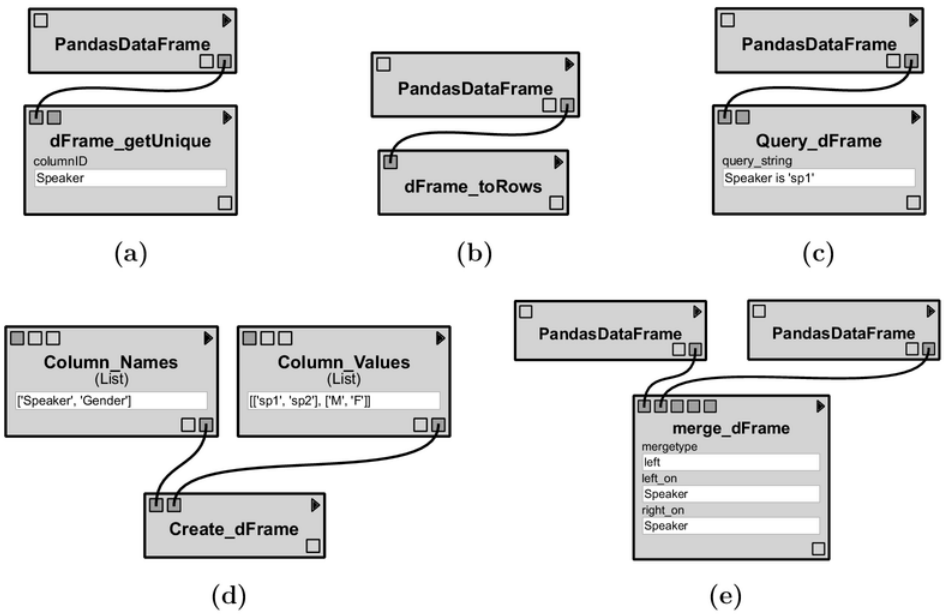
5.2.1 Implementation

This is the first attempt at building a speech analysis workflow in the VisTrails environment. In this stage, the goal is to construct the workflow in its entirety using only the default data and control-flow modules/packages that come shipped with VisTrails. As a result, this workflow would be built at the highest practical granularity that VisTrails allows. Observations can be made with regard to how effective this level of granularity is, usefulness of data structures, what are the benefits of various workflow characteristics (e.g. usability, maintainability, readability, speed, ...etc). From these observations, some insights can be made for the next implementation.

In this first iteration, Custom "python source" modules are only used to create modules for interfacing with data and speech analysis libraries, and not for encapsulating any major logical processes. Some of the necessary processes that these modules perform are provided in the following list. Apart from these processes listed, the major control-flow elements will be implemented using the default modules wherever possible, and will operate at the highest practical granularity.

- Interfacing with speech libraries —(reading/writing textgrids, querying text-grids for interval data)
- OS call (for R) — This is necessary for utilizing the wrassp formant tracking algorithm. This OS call executes an R script, which then runs the wrassp formant-tracking algorithm. The output is returned, and this is made into a python object.
- Basic Pandas utilities —(building DataFrames, pulling/writing rows, columns, basic querying)
- List operations — (zip, append, extend)

VisTrails comes shipped with a "table-data" library, but it was not utilized in this workflow. It provides some essential table operations (constructing tables, pulling column lists, very basic arithmetic queries), but it is limited in functionality compared to Pandas. Utilizing Pandas within the workspace was actually quite tricky. Working with Pandas generally happens very concisely within a few lines of native python code, but one of the goals for this task was to eliminate scripting entirely. This meant that some custom modules for interacting with Pandas data had to be constructed.



columnID: Speaker

query_string: Speaker is 'sp1'

['Speaker', 'Gender']

[['sp1', 'sp2'], ['M', 'F']]

mergetype: left
left_on: Speaker
right_on: Speaker

Figure 5.1: Pandas-Op Modules

Figure 5.1 shows some examples of some simple Pandas modules that were created for this workflow. The intention for these modules was to provide access to the essential Pandas operations through a simple, parameterized interface. This includes functionalities like:

- create DataFrame — construct a DataFrame from a list of column names, and a list of column values
- query DataFrame — return a filtered DataFrame that contains only samples that match the query string
- DataFrame to rows — return a list of all rows in the DataFrame. This list can be passed through VisTrails control-flow modules to iterate over the samples in a DataFrame
- get unique — return the unique values in a DataFrame column
- merge DataFrame — join two DataFrames together

Some of these implementations were more intuitive than others. The **merge** module (Figure 5.1e), for example, translated quite well to VisTrails. Figure 5.2.1 shows how this is traditionally accomplished using a script — the VisTrails version simply calls this function using the parameter values it has been assigned.

```
merged_dFrame = pd.merge(dFrame_left,
                         dFrame_right,
                         left_on=left_on_index,
                         right_on=right_on_index,
                         how=mergetype)
```

The "**create_dFrame**" module also translates very well to VisTrails. This single module can be used to very quickly construct a DataFrame within the workflow using only two parameters.

```
------------------------------------------------------------
column_names:        [A,          B,          C]
------------------------------------------------------------
colume_values:   [ [1,2,3],    [4,5,6],    [7,8,9] ]
------------------------------------------------------------
                     A            B          C
                     -------------------------
                     1            4          7
                     2            5          8
                     3            6          9
```

What does not translate very well to VisTrails are Pandas operations that can take functions as parameters. This can be very useful for tasks like "**query_dFrame**". Traditionally, Pandas can call some defined function in a query, but in VisTrails, this would require hard-coding a function in a python source module and then passing this as output to some target input port. There is no way to define a callable function *without* writing it as a script. Subworkflows can be constructed and can interface with VisTrails control-flow modules, but these are not the same as a python function, and cannot be called by Pandas. Consequently, the query_dFrame module is restricted to only Pandas.eval [18] (string expression) queries.

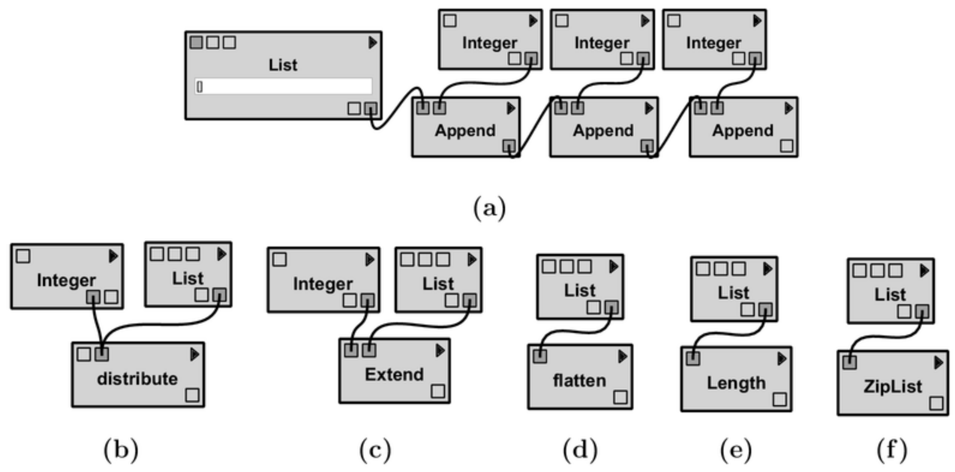


Figure 5.2: List-Op Modules

By default, VisTrails also doesn't include any modules for performing some fairly fundamental list operations. This includes operations such as:

- append
- length
- zip
- flatten
- get element at index

These had to be constructed for this workflow. Some examples of how these may be utilized within a workflow can be seen in Figure 5.2. The final set of modules was not entirely "comprehensive", but was sufficient to provide all of the necessary data-management functionalities for *this* workflow. Incorporating the more intricate functionalities of the Pandas package would require further investigation, but this falls outside the scope of this project.

The final workflow consisted of a total of 130 VisTrails modules. This can be seen in Figure 5.3. Subworkflow schematics for each of the core logical processes are also provided in appendix A2.

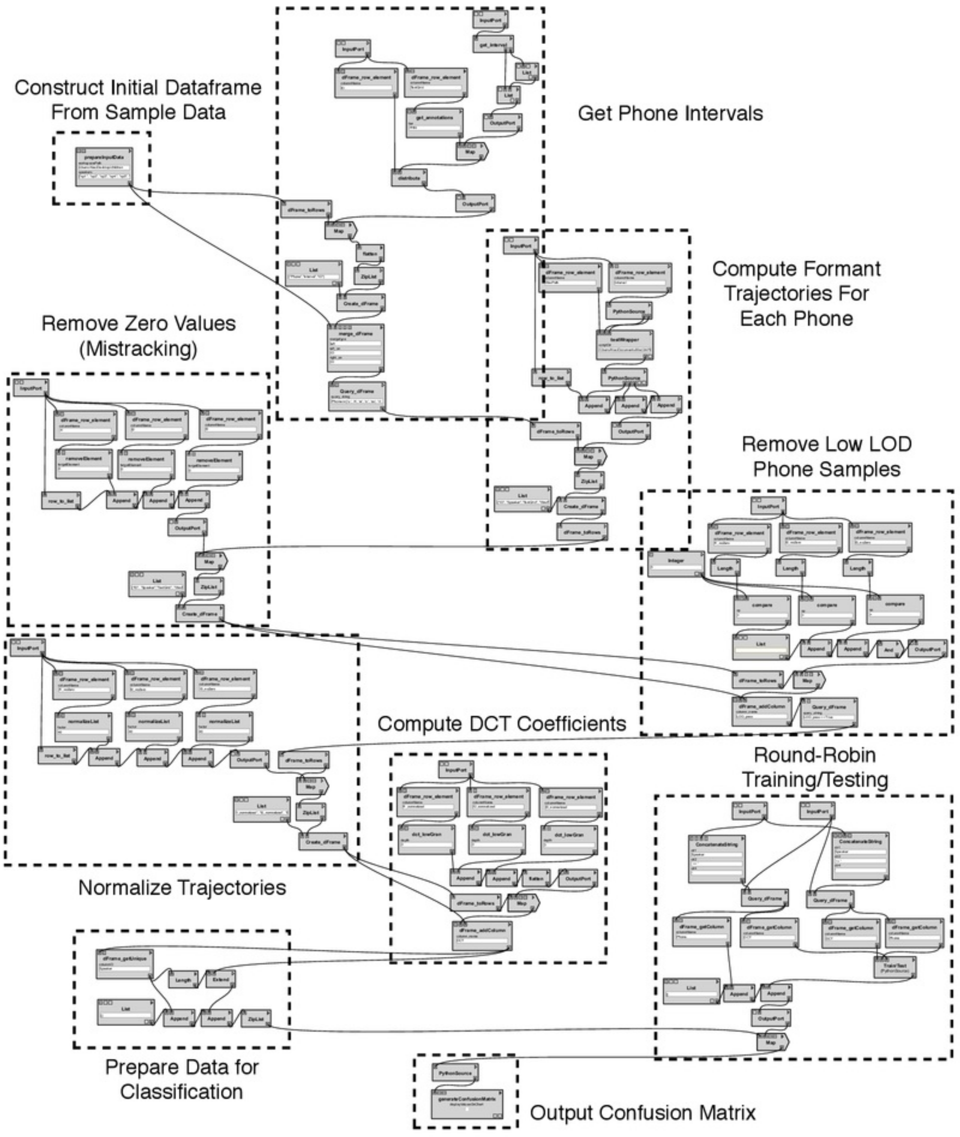

Construct Initial Dataframe From Sample Data
Get Phone Intervals
Compute Formant Trajectories For Each Phone
Remove Zero Values (Mistracking)
Remove Low LOD Phone Samples
Compute DCT Coefficients
Round-Robin Training/Testing
Normalize Trajectories
Prepare Data for Classification
Output Confusion Matrix

Figure 5.3: High Granularity Workflow

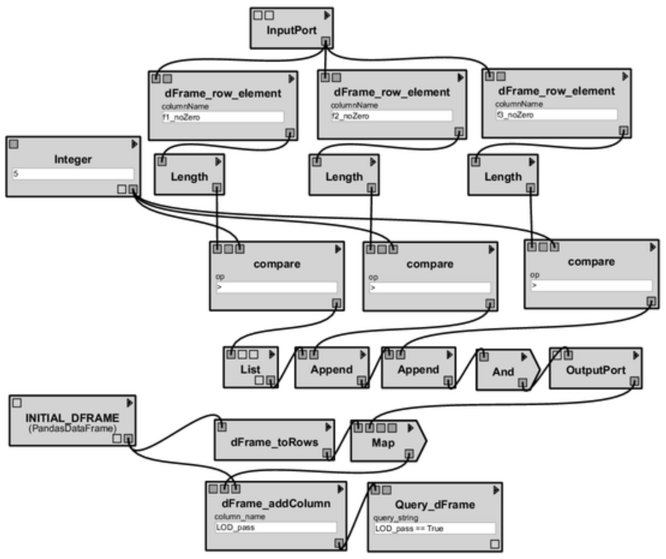


Figure 5.4: Remove Low LOD

The "Remove Low LOD" subworkflow illustrates one of the challenges that was mentioned earlier. Because it is not possible to pass a DataFrame query function using a Pandas.eval string (in this case, a function that checks if all three recorded formant measurement have a sample length greater than 5), this process is done iteratively instead. The resulting logical steps that make up this subworkflow (pictured in Figure 5.4) are as follows:

- **Step1:** The initial DataFrame is split into a list of it's component DataFrame rows
- **Step2:** Each row is passed through the mapped subworkflow, and an output list containing the results of all iterations is sent to the output port of the "map" module. For this subworkflow, the lengths of the three formants in a data-sample are checked to see if they pass the threshold length of "5". If *all* three formants pass the test, then the output for that row is true.
- **Step3:** The list of Boolean outputs are added as a column to the original DataFrame under the name "LOD_pass"
- **Step4:** *Now*, the DataFrame can be queried with a Pandas.eval string ("LOD_pass == True"). The output DataFrame contains only samples that pass the LOD threshold.

This was a re-implementation of the workflow completed in Jupyter Notebook. The resulting VisTrails output is, as expected, the same as the output from the Jupyter Notebook. At a glance, clearly the design space offered at such a granularity is quite large, and any number of possible workflows can be created with this set of modules. The question that arises is weather working at this granularity is at all practical. It took a lot of modules to construct the final workflow, and the mental gymnastics required for designing and debugging the program structure at this level was not trivial. Does this approach (at this granularity) hold any advantages at all over scripting in these areas, and would a lower granularity possibly offer a more practical solution?

5.2.2 Observations

There are two clear benefits of building workflows at this granularity.

- Transparency — Looking at the resulting workflow, although not concise, It's clear how the entire program is being executed. Nothing is being "hidden" or "abstracted away". Sharing workflow schematics at this granularity would be an effective way of demonstrating the fundamental workings of the entire program (high-level, and low-level)
- Design Freedom — Because this granularity utilizes modules that operate at such a low logical level, the design space that this toolset offers is extremely large. Although they might be difficult to debug or maintain, the construction of almost any complex workflow still remains possible

It's hard to quantify the difficulty of constructing this workflow without performing proper user-testing experiments, but what is clear is that it is not trivial. While the ability to entirely visually program a workflow removes the barrier-to-entry from users who may not have experience writing low-level source-code, many major complexities still remain. Much of the lower-level programming logic still has to be explicitly defined within the workflow, and this still (especially at this granularity) requires much of the same foundational knowledge of programming concepts. There are no major improvements to the maintainability, or the modifiability of the program. Clearly making a significant change to the original analysis task, or to any of the major logical processes, would require major refactoring. From this first implementation, it appears that a lower granularity solution would probably be much more practical.

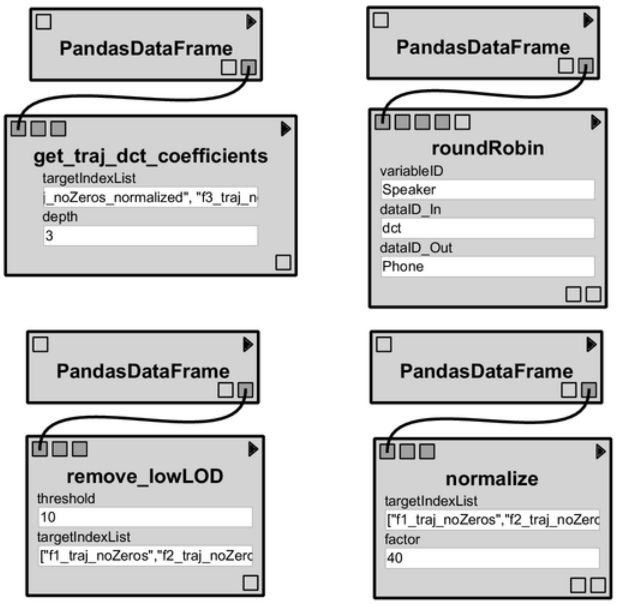
<u>Resulting Workflow:</u>

- Number of Modules: 130
- Execution Time: 35 minutes 24 seconds

5.3 Low Granularity Workflow:

5.3.1 Implementation

The next experiment was to implement the same VisTrails workflow from the previous section, but at a lower granularity. This will involve creating a collection of custom modules that are specialized to execute the higher-level logical processes performed in the workflow. Ideally these modules are not just useful for this analysis task, but are also applicable to a large number of other speech analysis tasks. Ultimately, the goal is to maintain the highest possible design space, but requiring the least number of modules to construct workflows.



get_traj_dct_coefficients
targetIndexList
j_noZeros_normalized", "f3_traj_n
depth
3

roundRobin
variableID
Speaker
dataID_In
dct
dataID_Out
Phone

remove_lowLOD
threshold
10
targetIndexList
["f1_traj_noZeros","f2_traj_noZerc

normalize
targetIndexList
["f1_traj_noZeros","f2_traj_noZerc
factor
40

Figure 5.5: Low-Granularity Modules

Shown above are some examples of the low-granularity modules created for this new workflow, and the execution parameters that can be specified. Because DataFrames take the role of "primary data-structure" in this toolset, simple and concise parameter options can be given to each of the modules. The "remove_lowLOD" module, for example, does with one module what took 17 modules in the higher granularity workflow, and requires only 2 input parameters. A user simply needs to know the DataFrame key(s) of the data they wish to operate over, and specify an integer for the threshold level-of-detail — (in this case, determined by the length of a list). The module then outputs a DataFrame that has filtered away any samples not above the threshold. The snippet found in Listing 5.1 shows the python code that makes up this module. Note that because this module is specialized for this sub-task using a hard-coded python script, the low_LOD samples can be filtered out efficiently in a single line of this script.

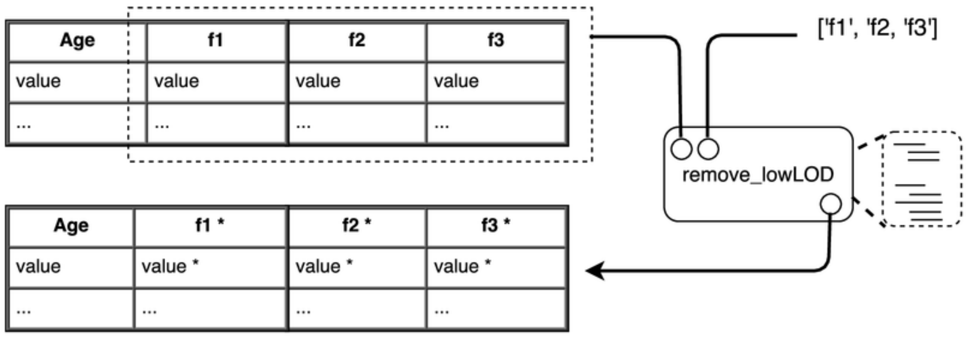


Figure 5.6: "remove_lowLOD" DataFrame Visualization

Listing 5.1: "remove_lowLOD" Python Code

```
class remove_lowLOD(Module):
    _input_ports = [IPort(name="dataFrame", signature="PandasDataFrame"),
                    IPort(name="threshold", signature="basic:Integer"),
                    IPort(name="targetIndexList", signature="basic:List")]
    _output_ports = [OPort(name="df_result", signature="PandasDataFrame")]

    def __init__(self):
        Module.__init__(self)

    def compute(self):
        df_result = self.get_input("dataFrame")
        for targetIndex in self.get_input("targetIndexList"):
            df_result = df_result[df_result[targetIndex].map(len) > self.get_input("threshold")]
        self.set_output("df_result", df)
```

Another example, Figure 5.2, shows the python-source for the 'roundRobin' module. In the high-granularity workflow, this process took 15 modules to construct, but now these operations have been hard-coded into a single unit, and with easy-to-configure parameters. The user doesn't need to concern themselves with the lower-level processes that make up the logic within the module, yet they can still utilize these functionalities to full effect. All the user needs to do is provide an input DataFrame, and specify which columns correspond to each of the core round-robin parameters (input-data, output-data, group-by).

Listing 5.2: “roundRobin” Python Code

```
class roundRobin(Module):
    _input_ports = [IPort(name="dataFrame", signature="PandasDataFrame"),
                    IPort(name="variableID", signature="basic:String"),
                    IPort(name="dataID_In", signature="basic:String"),
                    IPort(name="dataID_Out", signature="basic:String"),
                    IPort(name="classifierConstructor", signature="PandasDataFrame")]
    _output_ports = [OPort(name="results_byFactor", signature="basic:List"),
                     OPort(name="results_All", signature="basic:List")]

    def __init__(self):
        Module.__init__(self)

    def compute(self):
        dataFrame = self.get_input("dataFrame")
        factorID = self.get_input("variableID")
        dataIn = self.get_input("dataID_In")
        dataOut = self.get_input("dataID_Out")
        factorValues = dataFrame[factorID].unique().tolist()
        classifiers = [GaussianNB() for factor in factorValues]
        results = []
        for i in range(0,len(factorValues)):
            trainingData = [list(dataFrame[dataFrame[factorID] != factorValues[i]][dataIn]), list(dataFrame[dataFrame[
                 ↪ factorID] != factorValues[i]][dataOut])]
            testData = [list(dataFrame[dataFrame[factorID] == factorValues[i]][dataIn]), list(dataFrame[dataFrame[
                 ↪ factorID] == factorValues[i]][dataOut])]
            classifiers[i].fit(trainingData[0], trainingData[1])
            predictions = classifiers[i].predict(testData[0])
            results.append([predictions, testData[1]])

        allInputs = [item for sublist in [x[0] for x in results] for item in sublist]
        allOutputs = [item for sublist in [x[1] for x in results] for item in sublist]
        self.set_output("results_byFactor", results)
        self.set_output("results_All", [allInputs, allOutputs])
```

These modules are not bound to this particular analysis task. The round-robin module, for example, can be used for classifying experiments in an entirely different workflow, using entirely unrelated data-sets. It goes to show how properly utilizing a data-structure (like a Pandas DataFrame) to serve as a kind of universal data-protocol is a powerful way to make workflow modules more concise, interoperable, and extensible. The module design itself isn't the major factor, but rather, the uniform data-structure that ALL modules share that provides these desired characteristics.

The final workflow can be seen in Figure 5.7. It required a total of only 11 modules (as opposed to 130 modules in the previous implementation).

5.3.2 Observations

Compared to the previous workflow, this low-granularity workflow is a much more practical implementation, and it showcases some clear benefits of utilizing workflow systems within the domain of acoustic phonetics. Some advantages of working with speech-analysis modules at this granularity include:

Reduced complexity — The logical complexity of finalized workflows are greatly reduced by working at this level of abstraction. Starting from a collection of modules designed to operate at this granularity, implementing a data-analysis pipeline is as simple as consecutively ordering each major sub-task. Management of major control-flow elements, and low-level data access, is all handled within the module.

Computational efficiency — Execution of this workflow, using the same sample data from previous implementations, took only 19 minutes and 44 seconds. This is likely, largely, because of the overhead required for the execution of each VisTrails module. The computation of each module is cached, and logged, and requires extra steps to account for the execution of the scripted infrastructure that underlies each module. In a low-granularity workflow, most iterative processes occur within the module itself, and are not subject to such an overhead

Readability — While a higher granularity workflow may offer full transparency, this more abstracted implementation is much more clear and concise. Someone reading this workflow can easily see important logical steps throughout the analysis pipeline, and is not overwhelmed by extraneous control-flow elements that are of little concern

Modifiability — Making changes to the existing workflow, whether that be a fundamental change to the analysis task, or adding additional branches for further analysis is as simple as dragging in the appropriate module, and supplying the appropriate data parameters.

Easily configurable — Modules at this granularity, and workflows built with these modules, are easily configurable by default. As discussed earlier, all adjustable parameters can be modified through a simple interface on the module. In the previous workflow, unless the workflow has been well-engineered from the ground up, changing these parameters is not trivial, and likely spread throughout the workflow.

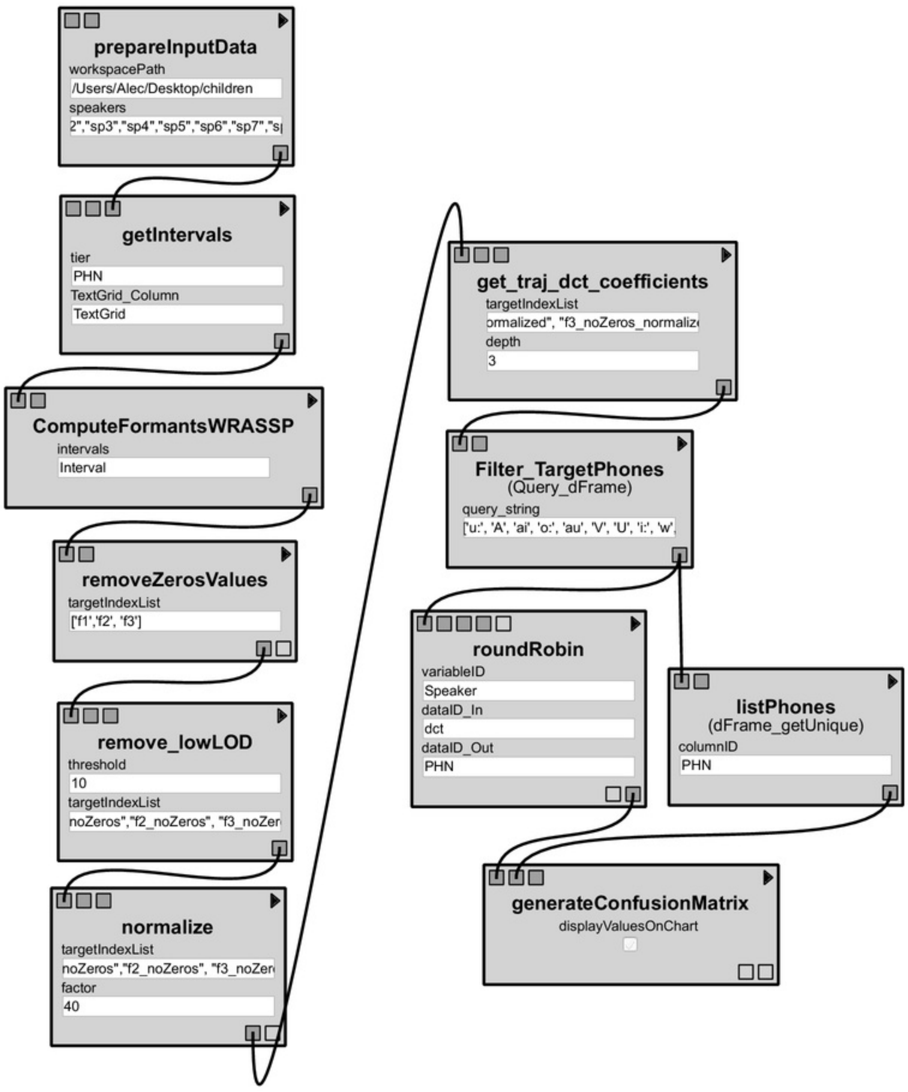
There are also some drawbacks when working at this granularity. This includes:

Reduced transparecy — While the readability of a workflow might be greatly improved by having modules represent only the high-level logical processes, the workflow gives no indication of what lower-level processes are actually occurring.

Reduced design space — There are some limiting factors when using modules at this granularity. A user gives up some control over the low-level logic happening within each module, and only has control over the external parameters of the module. This means that the number of possible workflows that can be constructed is restricted to the scope of the toolset. If the library of modules is quite large, then this increases the design space, but if a particular functionality is not offered, then a user is forced to implement it themselves by writing a python-source "patch", or construct a workaround using whatever modules already exist.

Resulting Workflow:

- Number of Modules: 11
- Execution Time: 19 minutes 44 seconds



prepareInputData
workspacePath
/Users/Alec/Desktop/children
speakers
2","sp3","sp4","sp5","sp6","sp7","s|

getIntervals
tier
PHN
TextGrid_Column
TextGrid

ComputeFormantsWRASSP
intervals
Interval

removeZerosValues
targetIndexList
['f1','f2', 'f3']

remove_lowLOD
threshold
10
targetIndexList
noZeros","f2_noZeros", "f3_noZer

normalize
targetIndexList
noZeros","f2_noZeros", "f3_noZer
factor
40

get_traj_dct_coefficients
targetIndexList
ormalized", "f3_noZeros_normaliz
depth
3

Filter_TargetPhones
(Query_dFrame)
query_string
['u:', 'A', 'ai', 'o:', 'au', 'V', 'U', 'i:', 'w',

roundRobin
variableID
Speaker
dataID_In
dct
dataID_Out
PHN

listPhones
(dFrame_getUnique)
columnID
PHN

generateConfusionMatrix
displayValuesOnChart

Figure 5.7: Low Granularity Workflow

Chapter 6

Conclusions and Future Work

6.1 Conclusions

After completing this project, it has become clear just how critical the underlying data-structure is to the construction of an effective workflow, and component modules. Because all data traveling through the pipeline is accessed purely through the input/output ports of workflow modules, choosing an effective vessel for packaging and accessing this data has a direct impact on the all aspects of the workflow, especially in terms of module interoperability, extensibility, and flexibility. The Pandas DataFrame object utilized in this project served as a kind of universal data-protocol that allowed all modules to be maximally interoperable at all granularities, and facilitated the smooth integration of various module parameter options. Because this Data-Structure is shared between all modules, future extensions and integrations with the tool-set can be made with minimal effort.

There are some clear trade-offs between implementing workflows using traditional scripting approaches, and using visual programming environments like VisTrails. An outline of some key observations relating to prominent workflow characteristics are listed below:

Transparency — Traditional scripted workflows win hands-down in terms of workflow transparency, as the entire execution of a program is explicitly defined in the workflow script. In visually programmed workflows, lower-level processes are hidden away to varying degrees depending on the level of abstraction used in the toolset. The lower the granularity of the workflow, the less transparency is available, as multiple intermediary processes are hidden within a single module.

Readability — What traditional scripts gain in transparency, they lose in readability. Visual workflows win in this arena, as they offer a more abstracted, much more familiar visual representation of workflow structure. Readability of visual workflows is also effected by workflow granularity. The clear-and-concise nature of the low-granularity workflows provide a more naturally coherent representation of the analysis task being performed, while tracing logic in low-granularity workflows is less immediately clear.

Design Space — The design space of a particular package/toolset depends on how comprehensive and dynamic its component modules are. It appears that the design space decreases with granularity, but realistically a package might feature a mixed-granularity toolset that reduces these imposed restrictions. Clearly, a purely scripted workflow is more versatile, but it is worth noting that VisTrails does allow the inclusion of hard-coded python-source modules if necessary.

Extensibility — There are two ways to assess extensibility. The first relates to the extensibility of specific workflows, and the second is extensibility of a package/toolset. Concern is primarily on the "ease" of extensibility, not the extent (all can be extended to an equal degree given enough resources). Extending a workflow in VisTrails, from the perspective of both a technical and non-technical user, is much more intuitive than with a scripted workflow. Given that a toolset is well-designed and comprehensive, it is as simple as dragging extra modules into the workspace and linking them together. If a user wanted to extend a traditionally scripted workflow, they might need to a reverse-engineer all parts of the script (and related libraries) in order to properly integrate new functionality. In a VisTrails workflow, low level integrations are of no concern to the user, as maximum compatibility is maintained by-design between each module.

As mentioned earlier, the ease of extending a VisTrails package is primarily dependent on one element — the shared data structure. If all modules share a kind of "protocol" for data representation, and passing this data between modules, then this allows any additional modules added to a package to be more easily compatible throughout the entire workflow.

Efficiency — A VisTrails workflow requires a lot of extraneous overhead/infrastructure for executing workflows. The assignment of input/output ports, the caching of module outputs, and the logging of module executions make working with *high*-granularity workflows less efficient. Low-granularity workflows execute most of control-flow operations within each module, so this is not felt as infrastructure-level overhead. Some simple timing analysis experiments (performed on the main analysis task implemented this project) made this distinction quite clear.

- High-granularity execution time: 35min 24sec
- Low-granularity execution time: 19min 44sec
- Jupyter Notebook execution time: 4min 30sec

Approachability — VisTrails workflows, at any granularity, are much more approachable from the standpoint of a non-technical user. The ability to program workflows without writing source-code removes a large barrier-to-entry for many of these users. After implementing VisTrails workflows at multiple granularities in this project, the clear winner in terms of approachability is the low-granularity workflow — simply because it eliminates the need for the user to construct computational process, and control-flow structures themselves.

In Summary:
When comparing granularities within the context of acoustic phonetics research, and how workflows can be applied to this domain with the goal of improving reproducibility and approachability of data-analysis tools, "low granularity" workflows seem to be the most applicable. This degree of abstraction narrows a users field of concern to the specific high-level tasks in an analysis pipeline, so the barrier-to-entry imposed by management of low-level logic is greatly reduced. In this environment, a user can modify the execution a sub-workflow task simply adjusting the associated module parameters, instead of needing to re-factor many different workflow modules throughout the schematic. This process is much more approachable to the average "non-technical" user. It's worth noting that if the fundamental data-structure is shared between toolsets of different granularities, in many instances these tools can still be used together. A situation where a low-granularity workflow might be restricting in design space could be "worked-around" by introducing some high granularity modules at these locations. Because granularity has such a large effect on various workflow characteristics, modules in a mixed-granularity toolset should be appropriately grouped so users can design workflows around this distinction accordingly.

This project has shown that VisTrails is definitely viable tool for improving accessibility to data analysis tasks, and improving research reproducibility in acoustic phonetics research. The paradigm lends itself very well to this type of research (particularly the low-granularity workflows), and it is possible to reproduce real-world acoustic phonetics analysis tasks in VisTrails using these packages of specialized domain-specific modules. Some of the many advantages provided are:

- Clear and concise workflows and workflow schematics
- Fast development times
- Highly extensible, shareable, and modifiable workflows
- Improved accessibility/approachability, and reduced barriers-to-entry

Of course, there are trade-offs that come with these advantages. One of the more obvious ones is the dependency on the VisTrails infrastructure for executing VisTrails workflows. For someone experienced writing scripts, implementing smaller analysis tasks in VisTrails might be less practical. Other drawbacks come in the form of minor performance issues when saving, switching loading, copy/pasting large workflows (a rare-sight in a text-editor). Also the lack of a large developer community makes it quite difficult to find solutions to specific issues.

6.2 Future Work

6.2.1 Building More Workflows

This project focused primarily on creating multiple implementations of only a single analysis task. Future work could include constructing workflows for a large number of analysis tasks, with special consideration placed on workflow adaptability through extensive parameter options. Perhaps with the goal of arriving at a suite of common workflows that can be quickly-and-easily adjusted (or extended) to fit the requirements of a particular analysis task.

6.2.2 Testing Other Features of VisTrails

This project surrounded the "workflow" components of VisTrails — e.g. building workflow modules, constructing workflows, and analyzing the trade-offs of various workflow design elements. This left many of the additional VisTrails features, and how they could be effectively utilized, somewhat overlooked. Some of these features could be investigated in future work.

- Outsorcing computational tasks to servers
- A closer look at the provenance system and its features
- Interactive data outputs

6.2.3 User Testing

All observations in this project related to "ease of use" and "complexity" were taken from first hand experience. Assumptions had to be made on how approachable a workflow might be from the perspective of a unexperienced user. But there is no substitute for organized user testing in assessing how intuitive a tool-set is to users with limited exposure.

User testing, in this case, might involve a more simplified rendition of the analysis task that was performed as part of this project. This could resemble the following scenarios:

<u>**Case 1:**</u>
Step 1: User is provided with an existing workflow implementation in VisTrails
Step 2: User is asked to modify this this workflow slightly to experiment with a mildly different approach to the analysis task using different parameters, rearranged modules, or additional modules
Step 3: Observations are made on how intuitively the user can accomplish these tasks, common sources of difficulty are recorded. User is asked for their feedback on the task, and what they thought was, or wasn't, challenging.

Case 2:
Step 1: User is assigned two speech analysis tasks to implement in VisTrails.
Step 2: User is provided with two VisTrails packages (consisting of a restricted set of modules) designed at different granularities.
Step 3: The user attempts, to the best of their ability, to implement the target workflows using the packages. They switch the package they start with for each task.
Step 4: Observations are made of the users actions, common avenues of difficulty or frustration are recorded. The user is also specifically asked for their input on what they thought about each workflow implementation.

6.2.4 A More "Comprehensive" set of Speech Analysis Tools

This project required that some custom modules be developed for the analysis task workflow — for the purposes of interfacing with data/speech libraries, and encapsulating major logical processes. However, outside the context of this analysis task, the toolset is incomplete. A possible avenue for future work would be to develop a complete set of tools for acoustic phonetics that would include a more extensive set of computation modules and any necessary improvements to the exploratory modules written for this project.

6.2.5 Testing Other Workflow Systems

Other workflow systems, like Galaxy, have been explored for use in building and executing acoustic phonetics workflows. VisTrails is only rendition of visually programmed workflow systems, and there are many others that have yet to be explored. Some of the observations gathered from building workflows in VisTrails could serve as a reference point when assessing if other systems might offer any major advantages for use in this domain.

6.2.6 Exploring Other Domains

This project was an exploration of VisTrails purely within the context of acoustic phonetics research. But there are numerous areas that could that could highly benefit from this type of abstracted data analysis pipeline. Put simply, if a domain might involve executing any kind of data-pipeline, it could potentially benefit from exploring the role systems like VisTrails might play in streamlining those tasks, and encouraging repeatable research — areas like astronomy, biology, meteorology, finance, etc...

Future explorations with VisTrails could, similarly to this project, involve taking an analysis task within another domain, and constructing VisTrails workflows, and associated packages. Borrowing ideas from the proof-of-concept packages created in this project, and applying them to a different fields could provide good insight into how well paradigm holds outside of the context of this project.

Chapter 7

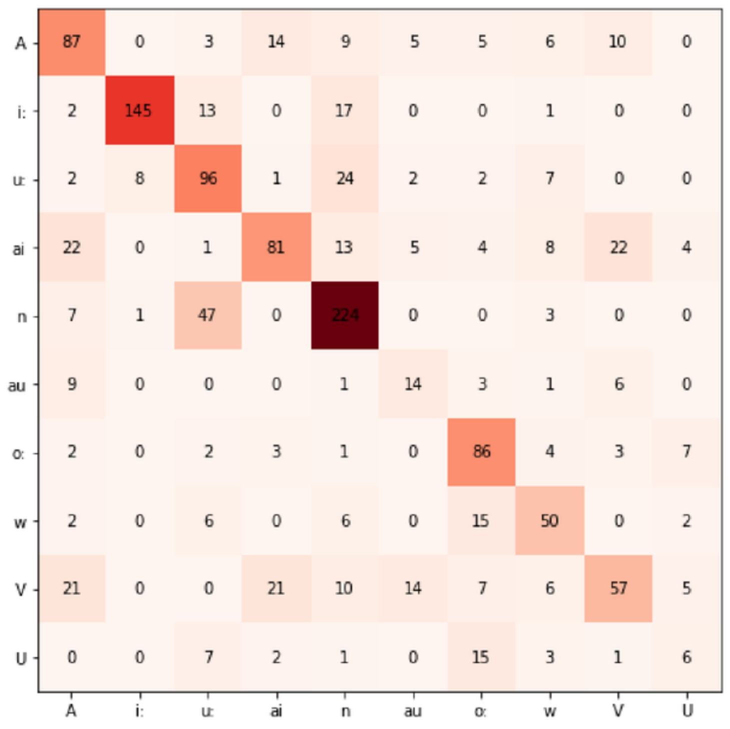
Abbreviations

DCT	Discrete Cosine Transform
FFT	Fast Fourier Transform
LOD	Level of Detail
UI	User Interface
XML	Extensible Markup Language (file format)
OS	Operating System

Appendix A

name of appendix A

A.1 Output Visualizations



	A	i:	u:	ai	n	au	o:	w	V	U
A	87	0	3	14	9	5	5	6	10	0
i:	2	145	13	0	17	0	0	1	0	0
u:	2	8	96	1	24	2	2	7	0	0
ai	22	0	1	81	13	5	4	8	22	4
n	7	1	47	0	224	0	0	3	0	0
au	9	0	0	0	1	14	3	1	6	0
o:	2	0	2	3	1	0	86	4	3	7
w	2	0	6	0	6	0	15	50	0	2
V	21	0	0	21	10	14	7	6	57	5
U	0	0	7	2	1	0	15	3	1	6

Figure A.1: Confusion Matrix

A.2 Subworkflows

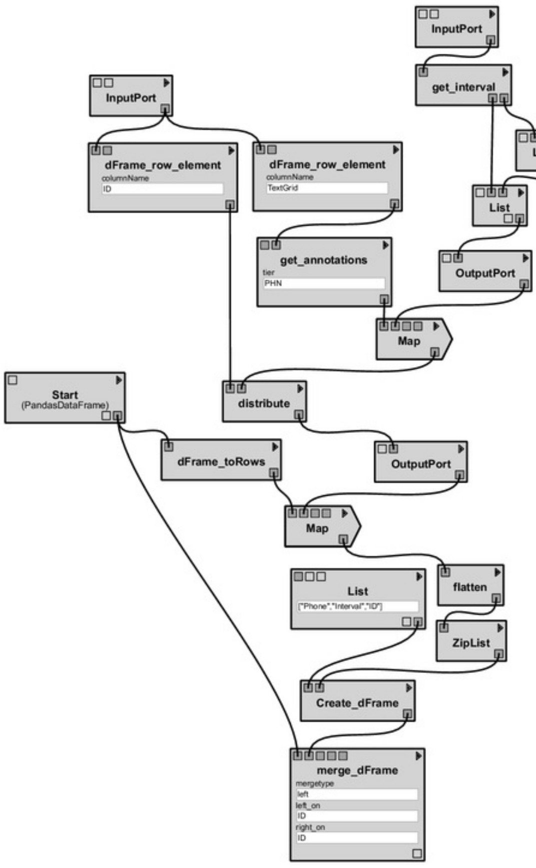


Figure A.2: Get Phone Intervals

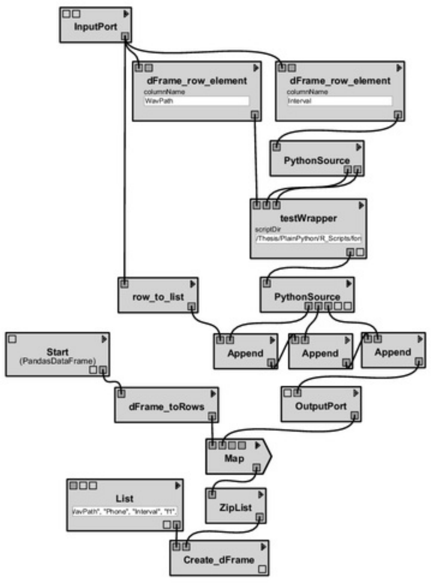


Figure A.3: Compute Formant Trajectories for Phones

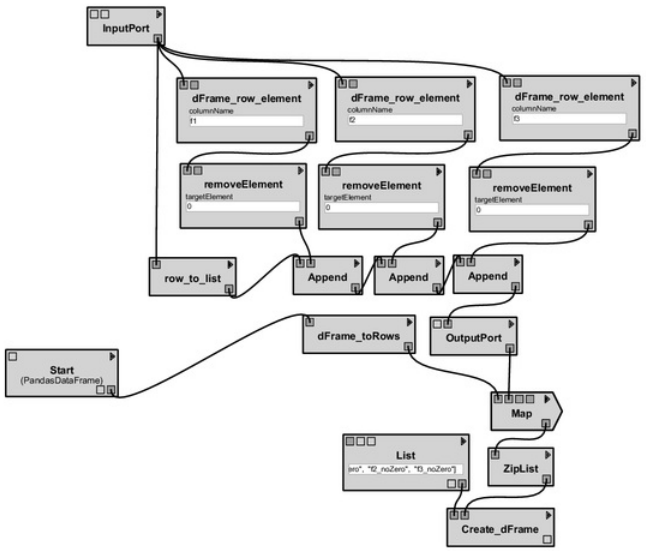


Figure A.4: Remove Zero Values

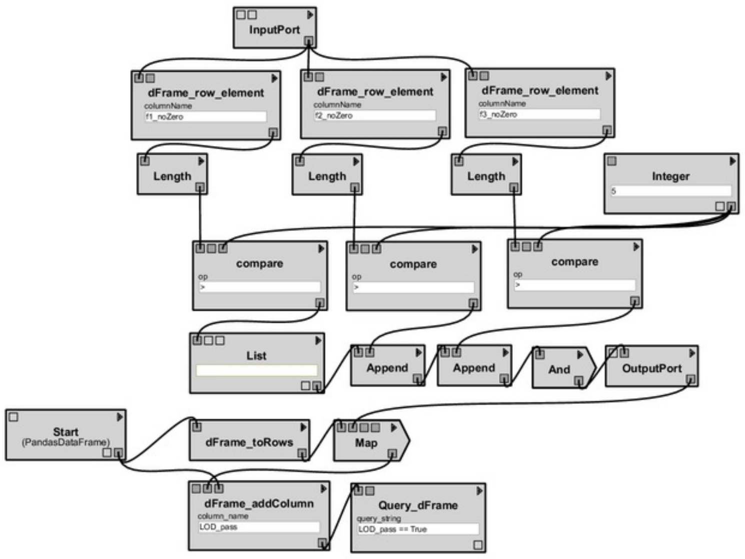


Figure A.5: Remove Low-LOD Samples

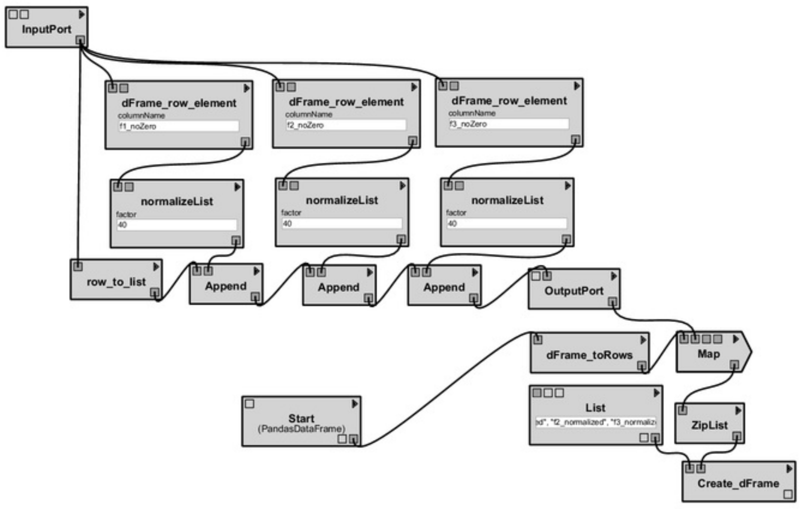


Figure A.6: Normalize Trajectories

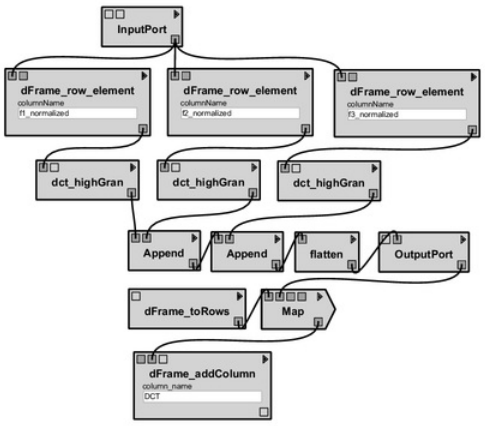


Figure A.7: Compute DCT Coefficients

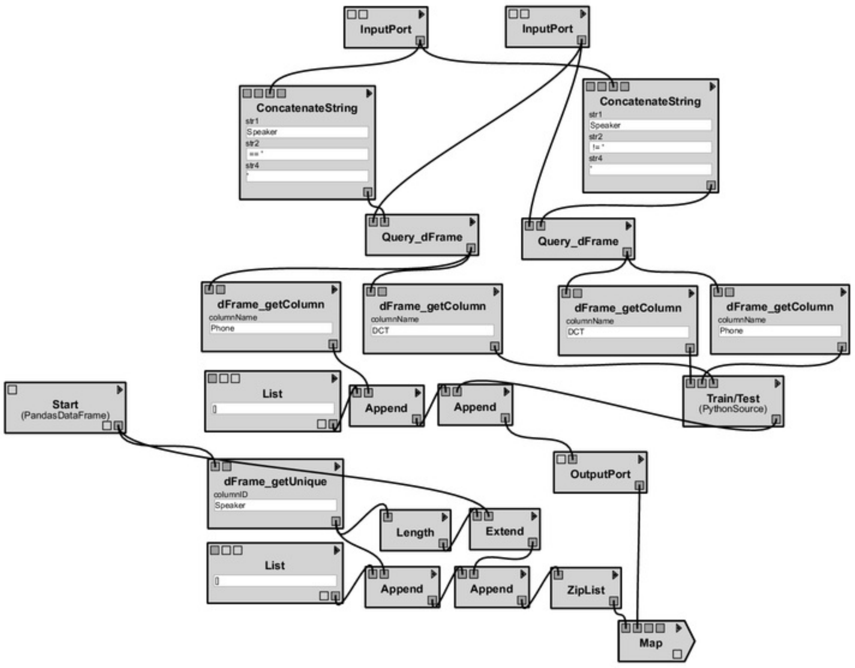


Figure A.8: Round-Robin Training/Testing

References

[1] Harrington, J. and Cassidy, S. (1999). Techniques in speech acoustics. Boston: Kluwer Academic Publishers.

[2] Andrews, W.D., Kohler, M.A. and Campbell, J.P., 2001, September. Phonetic speaker recognition. In INTERSPEECH (pp. 2517-2520).

[3] Clopper, C.G. and Pisoni, D.B., 2004. Some acoustic cues for the perceptual categorization of American English regional dialects. journal of phonetics, 32(1), pp.111-140.

[4] Bradlow, A.R., Torretta, G.M. and Pisoni, D.B., 1996. Intelligibility of normal speech I: Global and fine-grained acoustic-phonetic talker characteristics. Speech communication, 20(3-4), pp.255-272.

[5] Llisterri, J. (2017). *Speech analysis and transcription tools.* [online] Liceu.uab.es. Available at: http://liceu.uab.es/~joaquim/phonetics/fon_anal_acus/herram_anal_acus.html#Audiamus_N_Thieberger [Accessed 25 May 2017]. — licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

[6] Fon.hum.uva.nl. (2014). Scripting. [online] Available at: http://www.fon.hum.uva.nl/praat/manual/Scripting.html [Accessed 25 May 2017].

[7] Cassidy, S. and Estival, D., 2017. Supporting accessibility and reproducibility in language research in the Alveo virtual laboratory. Computer Speech & Language.

[8] Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P. and Bhagat, J., 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic acids research, 41(W1), pp.W557-W561.

[9] Romano, P., Bartocci, E., Bertolini, G., De Paoli, F., Marra, D., Mauri, G., Merelli, E. and Milanesi, L., 2007. Biowep: a workflow enactment portal for bioinformatics applications. BMC bioinformatics, 8(1), p.S19.

[10] Taverna.incubator.apache.org. (2016). Apache Taverna - Taverna in use. [online] Available at: https://taverna.incubator.apache.org/introduction/taverna-in-use/ [Accessed 25 May 2017].

[11] Afgan, E., Baker, D., Van den Beek, M., Blankenberg, D., Bouvier, D., ech, M., Chilton, J., Clements, D., Coraor, N., Eberhard, C. and Grning, B., 2016. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. Nucleic acids research, p.gkw343.

[12] Cassidy, S. (2016). A Galaxy Workflow for Acoustic Phonetic Analysis. [Blog] Available at: http://web.science.mq.edu.au/~cassidy/2016/10/18/a-galaxy-workflow-for-acoustic-phonetic-analysis/ [Accessed 25 May 2017].

[13] Koop, D., Freire, J. and Silva, C.T., 2013. Enabling reproducible science with VisTrails. arXiv preprint arXiv:1309.1784.

[14] Cassidy, S. and Watson, C., 1998, December. Dynamic features in children's vowels. In ICSLP.

[15] Software: Pandas Python Data Analysis Library. http://pandas.pydata.org/ (2017). Wes McKinney.

[16] Software: GameSalad. (2017). GameSalad. https://gamesalad.com/

[17] Griffis, J. Matthew (2013). Platform Studies: GameSalad. [pdf] pp.34-35. Available at: http://www.jmatthewgriffis.com/writing/Griffis_Game_Studies_Research_Paper.pdf

[18] Pandas.pydata.org. (2017). pandas.eval pandas 0.21.0 documentation. [online] Available at: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.eval.html#pandas.eval

[19] Winkelmann, Raphael., WRASSP, Github Repository, https://github.com/IPS-LMU/wrassp

[20] Buschmeier, Hendrik., TGT, Github Repository, https://github.com/hbuschme/TextGridTools/

[21] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), pp.2825-2830.

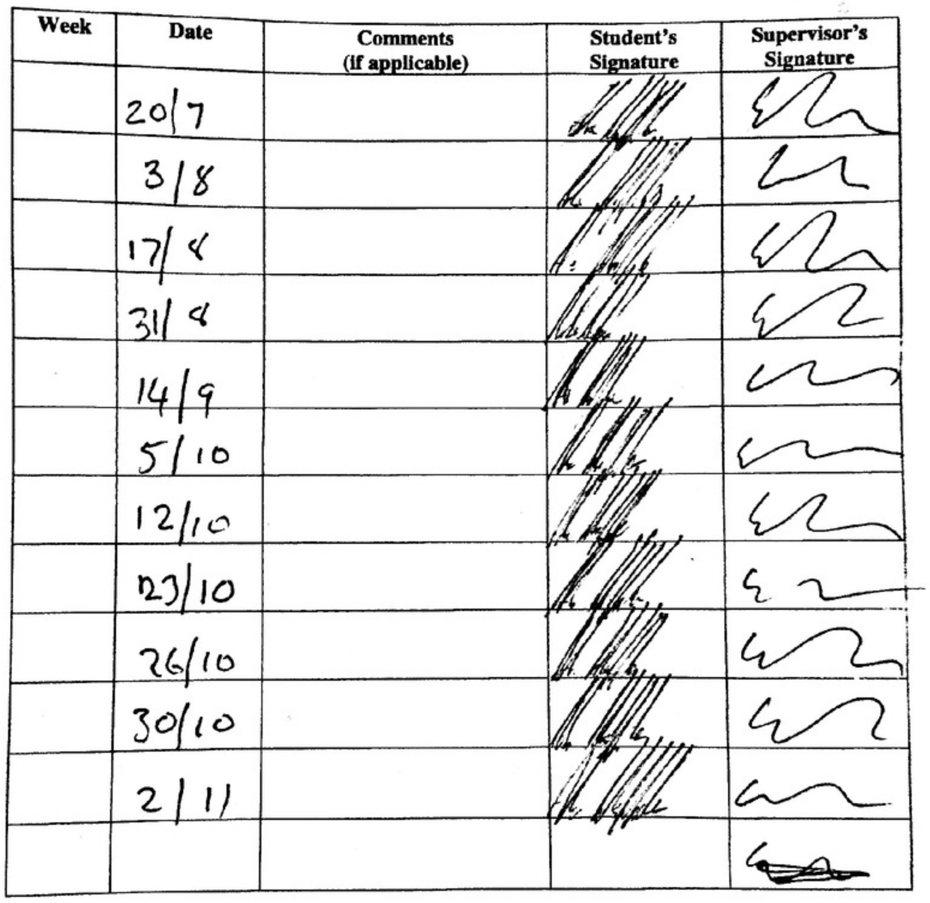
[22] Walt, S.V.D., Colbert, S.C. and Varoquaux, G., 2011. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2), pp.22-30.

[23] Jones, E., Oliphant, T. and Peterson, P., 2014. SciPy: open source scientific tools for Python.

[24] Kluyver, T., Ragan-Kelley, B., Prez, F., Granger, B.E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J.B., Grout, J., Corlay, S. and Ivanov, P., 2016, May. Jupyter Notebooks-a publishing format for reproducible computational workflows. In ELPUB (pp. 87-90).

[25] Scheuffele, Alec D., macspeech_vistrailsacousticphoneticsworkflows, (2017), Bitbucket repository, https://bitbucket.org/AlecScheuffele/macspeech_vistrailsacousticphoneticsworkflows

Consultation Log



Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
	20/7		[illegible]	[illegible]
	3/8		[illegible]	[illegible]
	17/8		[illegible]	[illegible]
	31/8		[illegible]	[illegible]
	14/9		[illegible]	[illegible]
	5/10		[illegible]	[illegible]
	12/10		[illegible]	[illegible]
	23/10		[illegible]	[illegible]
	26/10		[illegible]	[illegible]
	30/10		[illegible]	[illegible]
	2/11		[illegible]	[illegible]
				[illegible]