Automatic Biological Protocol and Python Code Generation using Human Speech

A thesis submitted by

Kevin Kapner

In partial fulfillment of the requirements for the degree of

Master of Science

In

Biomedical Engineering

Tufts University

May 2019

Adviser: David Kaplan

Abstract

Laboratory automation has become increasingly prevalent in biotechnology companies and pharmaceutical companies, allowing for thousands of experiments to be carried out weekly. In smaller scale research laboratories, the initial cost and learning curve to operating new machinery is often too steep to be of practical value. In this work, a novel method for protocol generation on the Opentrons laboratory robotics platform solely using verbal instruction was developed. Through the classification of the instructions groups using machine learning, the system is able to apply specific parsing methods that take advantage of unique characteristics that are typical of a given class of instructions. Analysis of the instructional phrases using natural language processing techniques yielded accurate and reliable code generation, even with varying sentence structures. This speech processing pipeline extends the usability of the Opentrons robotics platform to those with limited programming experience and makes the performance of common laboratory tasks more accessible.

Acknowledgements

I would first like to thank Professor David Kaplan for his long-term guidance over my entire undergraduate and now graduate career. The experiences I have had in his neuro group during the past four years have shaped how I view science and the skills and knowledge I have gained are invaluable. I also would like to thank Professor Soha Hassoun, for her unceasing encouragement and support for me and my exploration into the realm of computer science, which is something that I will be continuing in my professional career. I would like to extend another thank you to Professor Madeleine Oudin who has provided me with valued insights and suggestions of features for incorporation into this project from the perspective of a potential user, as well as her encouragement and support of the project from its initial presentation.

In the non-academic world, none of this would have been possible had I not experienced such an outpouring of support, encouragement, and love from my family and friends. I would like to especially thank my mom, who has put up with way too many phone calls at random hours of the day, and my dad, whose random texts telling me he is proud make me smile from ear to ear. I truly am lucky. I cannot express enough gratitude to my wonderful girlfriend Sidney; whose constant reminders of how much I am capable of have helped me beyond words. Her work ethic and compassion are something I try to emulate on a daily basis.

Table of Contents

List of Tables

List of Figures

Introduction

1.1 Background on Robotics in a Laboratory Setting

Laboratory robotics have become an integral part of modern science and are crucial to many biology-based industries such as pharmaceutical companies and biotechnology companies. Pharmaceutical companies can now scan thousands of compounds per week for therapeutic potential and work has been done on robots that are capable of generating and testing their own scientific hypotheses given a knowledgebase.[1,2] Robotics allow repetitive laboratory tasks, such as media changes and common assays, to be performed consistently and easily, freeing the researcher's time for more significant laboratory tasks, data analysis, and the design of new experiments. Furthermore, a key component of verifying scientific progress is the reproducibility of experimental results, ensuring that data is accurate and applicable to the larger question at hand instead of a specific microenvironment in a laboratory.

Reproducibility of the science that is published in journals is of major concern to labs that base future experimental designs and research questions on published literature, which is to say, all science labs. If a new study uses results based on a previously published study, which turns out to be irreproducible or false, then the validity of the new study, no matter how convincing, is subject to questioning. In the area of cancer biology, scientists at Amgen found that out of a selected fifty three landmark studies in the field, only six, or 11%, were reproducible.[3] Furthermore, even in journals with a high impact factor, over half of the non-reproducible studies that were examined were cited in secondary works.[3] This reproducibility issue can be combated systematically through a stricter vetting of journal articles, as well as a greater acceptance in the scientific community to publish negative results. Additionally, individual labs generating the data can take steps to increase the reproducibility of their results. A main way to complete this task is

to have a robotics platform, capable of logging its steps, carry out many of the common tasks.[4] Through automation, consistency of experiments can be obtained and the typical reliance on one technician to perform a single task to enhance reproducibility can be removed. Laboratory robotics can also reduce the chance of human exposure to toxic chemicals and increase the overall number of experiments performed by performing many labor intensive methods in a shorter time span or through the multiplexing of protocols.[5]

1.2 The Opentrons Platform and Project Goal

Opentrons is a laboratory robotics company based in Brooklyn, New York that specializes in relatively low cost, user friendly, and completely open-source hardware and software. They aim to "make robots for biologists" and want to allow scientists to "spend their time designing experiments and analyzing data."[6] They currently offer an OT-1 and an OT-2 model of their laboratory benchtop liquid handlers, along with extra hardware modules that allow for heating, cooling, and magnetic bead separation. The robots are relatively inexpensive in terms of other laboratory robotic platforms, costing $4,000 for their newest OT-2 model. To aid in protocol development for their machines, they have created an open-source Python package that comes prebuilt with many of their common containers, the company's name for things that fit into the slots on the robot bed, such as tip racks, 96 well plates, and small tube holders. Once imported, the user can begin coding the protocol in Python, or submit a protocol to Opentrons to have one of their developers create a protocol for you.

Once a protocol is developed, the user can upload it the Opentrons application that is available for Linux, Mac, and Windows machines and the application then guides the user through initializing all of the container objects and pipettes to ensure accuracy and reduce the chance of mechanical errors. In the newest version of the application, which works with the OT-2 model

only, they have implanted a beta version of a guided user interface, referred to as a GUI, designer for simple protocols. This GUI will allow users to skip the coding step and implement basic protocols, such as making a dilution, without the knowledge of Python. Unfortunately, it is not suitable for more complex tasks and clicking around in an interface will become unmanageable for more complex protocols that many have twenty or more detailed steps.[7] This project focuses on this issue: scaling up from basic protocols to more complex protocols. By developing a method to generate protocols using voice, one of the most natural form of human expression, the accessibility to users with no previous technical or computer science background expands and allows for faster, more accurate, and a more natural experience for the user. Just as a researcher would instruct a fellow lab member on how to perform an assay, the researcher can tell the robot how to do it in the same way and have a completely reproducible and shareable protocol from voice alone.

1.3 Project Outline

This project was broken into four separate parts to allow for a systematic and modular approach to the problem, beginning at the conversion of spoken audio into text and resulting in a GUI that complements the previously created Opentrons application.

1.3.1 *Voice-To-Text*

In this section, the logistics of the speech to text transcription are detailed. The user directly records instructions through either a built-in microphone or a set up external microphone and this module submits the corresponding audio file to the Google Cloud Speech-to-Text service. This service then returns a written transcription of the audio along with a confidence score, indicating how confident the service is in its transcription. Once the audio is transcribed into text, the text

can then enter the processing stages where the information pertaining to the protocol instruction is extracted and placed into the corresponding Python commands.

1.3.2 Text Categorization

The first step in translating the transcribed text into Python code is to classify the statement as belonging to one of five, broad, Opentrons categories. These categories were chosen because they represent the five main actions that the robotics platform is capable of performing and all Python commands in protocols can be categorized into one of the groups, which are: initialization, transfer, distribute, mix, and wait. An initialization statement sets up a container object in a specific slot on the robot bed, names it, and defines any other characteristics about that object as it pertains to the experiment. A transfer statement is defined as a liquid transfer where the pipette tip is discarded after one use. A distribute statement is defined as a liquid transfer where the pipette tip is not discarded after one use and can be used to "distribute" more liquid to other wells. A mix statement instructs the robot to mix the previously moved liquid after depositing the sample. A wait statement instructs the robot to pause for the specified amount of time, allowing for incubation of the sample or a delayed start of a next step. The categorizations of a statement are performed using a Naïve Bayes model that was trained on a dataset generated by eight volunteers who each provided step by step transcriptions of four protocols.

1.3.3 Text to Code

Once categorized, the statements are then parsed using category specific parsers. These parsers are designed to create the corresponding Python commands after extracting the required information from the instruction. Some parsers rely on a part of speech tagger, which is a natural language processing tool that labels each word in the sentence with its corresponding part of speech, while others rely on the existence of certain patterns in a word, such as a slot location

being identified as a letter immediately followed by a number, like A1. Furthermore, the initialize parser also uses another Naïve Bayes model to identify which Opentrons container should be selected when setting up the location. Each of these parsers are only called when an instruction is classified into its corresponding category. This saves computational time and allows for modularity, making it easy to edit pre-existing parsers or add a new one without affecting the functioning of any others.

1.3.4 Guided User Interface

The user interface combines all of the previous sections into a user-friendly application and provides an intuitive way for the user to create a protocol through voice. This GUI allows the user to re-record steps, view the current protocol up to the currently displayed step, navigate backwards and forwards in the protocol, and completely compile and generate a Python script that can be submitted to the Opentrons application for running. This GUI displays the transcription of what was said to the user, allowing for immediate correction if the transcription is incorrect. Furthermore, upon generation of the protocol, if there were any issues in extracting the information, Python comments are inserted in the unfilled areas, clearly instructing the user on how to fill them in. This serves as a fail-safe in case the parsers were unable to extract the proper information for one specific instruction, even though the rest of the protocol may be correct. The GUI is fully functional, and all of the language processing features and protocol navigation can be performed directly through the GUI itself.

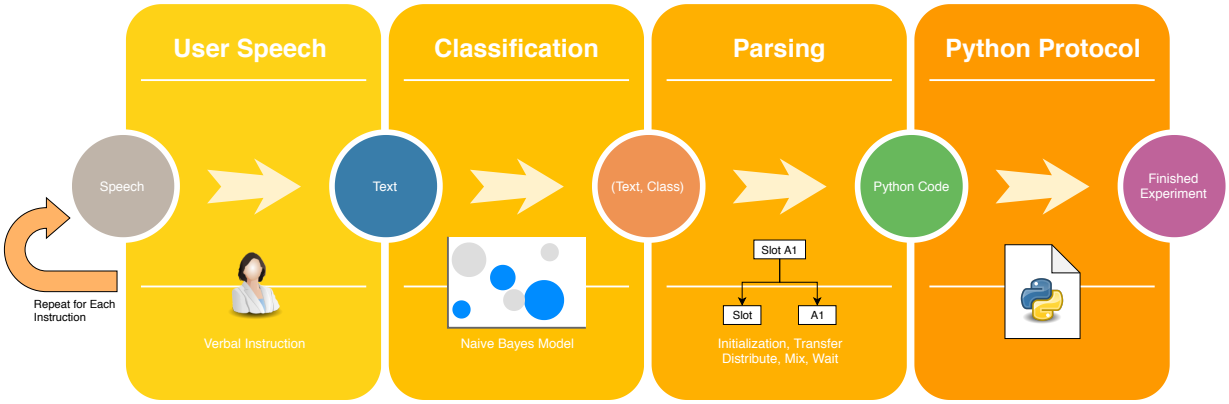

Figure 1. Outline of Analysis Pipeline
This schematic illustrates the analysis pipeline that a user's speech flows through when generating the protocol. The verbal instruction is classified, then parsed, then put into a Python protocol that can be run on the Opentrons platform.

1.4 Literature Review and Similar Work

Natural language processing has blossomed from its conception as a field in the 1950s, with the first major step in machine translation occurring in 1954 in the famous Georgetown Experiment where a small set of Russian sentences were translated into English and this is often cited as the first demonstration of computational language capabilities.[8] Starting in the 1990s, researchers in natural language processing, referred to as NLP, began to take advantage of large datasets of natural language introduced by the Linguistic Data Consortium in 1992.[9] Through manual annotation of this data, and text data slowly being added to the internet, statistical models were developed that were capable of tagging a word with its part of speech, referred to as POS tagging, and predicting sentiments of sentences, such as positive or negative meanings. In addition to these two fields, there's also higher-level functions such as naming entities, which is classifying a word into a category such as person, date, or animal, and dependency linking, which shows the linguistic linkages between all the related words in the sentence.[10]

As NLP problems have grown and become of increasing interest to researchers, indicated by the high prevalence of voice activated software that has become ubiquitous in the technology

field, such as the Amazon Alexa or Apple's Siri, a vast number of open-source tools have been developed. One of the leaders in the NLP field is the NLP Group at Stanford University, who have developed the Stanford CoreNLP NLP Toolkit, which comes with pre-trained models for POS tagging, named-entity recognition, and dependency graphing.[11] The NLP Group at Stanford also has developed an advanced sentiment analyzer, named SEMPRE, which is capable of transforming internet queries into a logical language that can query a database, with a 3% absolute improvement over the previous top performing method.[12] Furthermore, public libraries for languages such as Python have been created that take advantage of the CoreNLP tools and provide easy to use wrappers that allow for rapid natural language processing with state-of-the-art methods in only a few lines of code.[13]

Applications of these NLP techniques to robotics have ranged from aiding the elderly or persons with disabilities to open heart surgery.[14,15] In these situations, voice allows the user to move the mouse on the screen, move an entire robotic arm and instruct it to make precise cuts into a patient, or assist in controlling other equipment such as cameras.[16] There has been significant research in improving and demonstrating voice control of robotics, particularly in association with people with disabilities who may need to control a motorized wheelchair. Methods have been developed to construct the surrounding environment from speech, in conjunction with cameras, allowing for intuitive instructions such as "Turn down the second hallway after the kitchen."[17,18] These approaches are extremely versatile, as required by necessity, but in the research environment, the space of possible phrases is much smaller, allowing for a less complex and more specific approach. In research and factory settings, voice controlled robotics have allowed for automatic welding machines and vocal joysticks, with new methodologies developed that are capable of using the tone and duration of notes to add another dimension to the voice control

capabilities.[15,19] Even more, further analysis has also been done investigating the current lack of major incorporation of these technologies into the industrial world, and approaches suggested by such studies has been reviewed and kept in mind throughout the course of this project.[20]

Focusing more narrowly into the field of laboratory robotics, there has been an increase in the inclusion of automatic liquid handlers, as they are capable of performing labor intensive tasks accurately and quickly.[2] Unfortunately, these robotics platforms, being used in highly academic and research oriented environments, rarely incorporate any language functionality. This project aims to fill that gap, allowing for the accessibility and positive human factor elements of voice control to assist researchers in the lab to create protocols faster and without the learning curve and code testing that is required when hard-coding in a protocol for the Opentrons platform. This project most closely resembles that of Matuszek et al., who explored the potential of controlling a self-locomotive robot by providing it verbal instruction as one would to a fellow person, including complex directions with multiple steps in one sentence. Their models parse the natural language input and convert it into a robot control language, that can then guide the robot through a map of obstacles, similar to how this project converts natural language into Python commands.[21] Compared to their approach, this project implements a pre-mapping stage where instructions are classified before being parsed and there are differences on the methodology of parsing. The layer of classification aids in the accuracy and future extensions of the project, which is almost guaranteed by the constant incorporation of new features into the Opentrons robotics platform. With this new approach, the conversion from speech to code is more intuitive and easily upgradable, with multiple methods of parsing capable of being implemented in parallel.

Voice to Text Platform

2.1 Purpose and Goal

The first step in the transformation of speech to code is to create a transcription of what was said. From this written transcript, all of the post-processing and semantic understanding takes place to extract the necessary information. It then becomes critical to the success of the majority of the project to have a highly accurate transcription of the verbal instruction. To achieve this high level of accuracy, the method of transcribing the speech is outsourced to service providers who have developed extremely large scale and robust methods for speech to text conversion. The methods and skills that would be required to implement a highly efficient and accurate speech to text model locally would be tremendous. Technology giants such as Google and IBM already have successful methods and also have the vast computing resources necessary to process the data in a timely manner, thus this portion of the project focused on the decision of a service provider and not actually developing the computer science methodology to perform the transcription. Although there are inherent drawbacks in this approach, such as a lack of ability of fine tune the model to the specifications of this project or to see the architecture of how the model works, the benefits of using an already built and tested model allow for faster and more accurate project development, as well as more time to focus on the unsolved problem of going from natural language to code.

2.2 Choosing a Cloud Platform

With the advent of technology giants such as Amazon, IBM, and Google, and an increasing demand for high performance technology and data processing pipelines, many hard machine learning problems and techniques have rentable solutions. On a local scale, these problems require massive amounts of training data, in addition to advanced theoretical and practical knowledge, and state-of-the-art hardware. The problem of accurately capturing speech is far from trivial and is at

the center of many extremely popular technological devices such as Apple's Siri, Amazon's Alexa, and Google's Home. The mathematical and computational methods for voice transcription are typically grounded in probability and statistics, such as the case with hidden Markov models and neural networks.[22] For this project, the two models selected were the rentable cloud platforms provided by IBM and Google, which both offer popular and fully trained models for voice to text conversion that are trained on millions of speech samples and are run on the company's cloud computing servers.

Outsourcing the computation and model architecture design to professionals allows for a more accurate, and faster model that can provide all of the necessary transcriptions. It also allows for expansion of the project should more resources be needed. The downside to this approach is the lack of control in the training process and the inability to train the model further on application specific data, which could improve overall accuracy and usability.

2.2.1 Google

2.2.1.1 Capabilities

The Google Cloud Speech-to-Text service is part of the Google Cloud Platform and is offered through their AI and machine learning products.[23] At the time of implementation, this platform is capable of transcribing 120 different languages to text, in addition to recognizing up to five speakers in a given audio sample. The model is neural network based, however the logistics of the implementation are hidden from the public eye, so alteration or analysis of their methods is not possible. In conjunction with the actual model, Google provides an API that is accessible by installing the google-cloud-speech library for Python, which has extensive documentation.[23] These features, coupled with the documentation, make the Google platform ideal for this project, as it is easily scalable and allows for greater readability and modification of the source code for the project.

2.2.1.2 Price

The Google Cloud Speech-to-Text model has a tiered pricing scheme, where speech clips are billed in increments of 15 seconds. The billing for the model is on a monthly basis and each 15 second segment in the first 60 minutes a month are free. Following this free hour, the price becomes $0.006 USD per 15 second segment, up to 1 million minutes.[23] Audio submissions are rounded up, such that 15.1 seconds is billed as 30 seconds and anything less than 15 seconds is billed as 15 seconds. Using this pricing model, assuming that the first free 60 minutes were used up, a protocol consisting of 1000 voice commands, each of length 15 seconds, would cost $6. Once generated, a protocol can be used indefinitely, so most costs would be incurred in the early use of the platform and anytime new protocols were being added. At the scale of one lab, this price model makes operation costs of the platform negligible. Additionally, by opening up an account with the Google Cloud service, the user receives a $300 credit. This essentially makes the whole transcription process free, requiring over 208 hours of audio transcription post the free 60 minutes each month to use up this initial credit.

2.2.2 IBM

2.2.2.1 Capabilities

The IBM Speech to Text platform, like Google's, is an online service that connects to IBM's Watson, which provides all of the audio transcription and returns the text along with a confidence score.[24] This model allows for transcription form seven languages and also allows for a customizable model, which can be trained to recognize more specific use oriented language. IBM also provides an API, however interaction with this API has to be done via the Python requests library, as there is not a prebuilt Python package that handles API interactions. Similar to Google's service, there is extensive documentation on how to interact with the API and obtain transcriptions of audio files.

2.2.2.2 Price

The pricing model for IBM's service also follows a tiered pricing model. Their standard service model also runs on a monthly basis begins pricing at $0.02 USD for minutes 1 – 250,000 and then pricing drops to $0.015 USD for minutes 250,001 – 500,000, dropping again to $0.0125 for minutes 500,001 – 1,00,000 and final the lowest cost being $0.01 USD for any minutes 1,000,001 and above.[24] To use the customizable service, the first 1000 minutes are free each month, with an additional $0.03 USD per minute after that. The only credit they offer is to use the free "Lite Plan," where a free 100 minutes is awarded to you, corresponding to total savings of $2 USD.

Table 1. Comparison of Google and IBM Speech-to-Text Services
*Cost is assumed on typical usage defined as follows: 5 researchers, each recording 120 voice commands per week, over an average month of 4 weeks.
**The initial $300 credit from Google would require roughly 140 months at this usage level to begin costing anything. When it does begin costing money after the credit is exhausted, the cost will be $2.15 a month.

	Google Cloud Speech-To-Text	IBM
Cost*	Free**	$7.20
Number of Languages	120	7
API Interaction	Prebuilt Python library	Via Python requests library
Scientific Language Support Without Model Customization	Yes	No

2.2.3 Model Selection

When comparing the two models from a purely financial perspective, Google's service appears to be the better option given the $300 credit. Although IBM's model is slightly cheaper, at $0.02 per minute compared with $0.024 per minute with Google's model, on the scale of one laboratory, the pricing difference does not seem significant. Furthermore, if the customizable option were to be implemented with the IBM platform, then the small price advantage drops away, leaving Google's to be the cheaper one after the first 1000 minutes.[24] In terms of capability, the Google model is superior, recognizing far more languages and being able to distinguish between up to five individual voices. The prebuilt API package is also a plus and requires less set up. During preliminary testing, the IBM platform seemed unable to convert scientific phrases such as "microliter" or "pipette," transcribing them as "may her leader" and "pipe" despite intentional enunciation. The Google model had no such errors and did not produce any differences from the spoken speech when transcribing during testing where sample instructions from the volunteers were recorded and submitted for processing. With all of this considerations, the Google model was selected as the better option for this project and thus all of the speech to text conversion was done through submission to the Google Cloud Speech-to-Text service.

2.3 Organization of Speech Transcription Module

The python module developed for this subsection of the project is named Voice_to_Text and is further broken down into three sub-files, Voice_to_Text.py, VoiceConversion.py, and VoiceInput.py. This module provides all the necessary code to record an audio sample for a specified length of time, submit the audio sample to Google's Cloud Speech-to-Text service, and receive and output a text transcription, with a confidence score, of what was said.

2.3.1 Voice Input

The code in this file provides the functions necessary to take in audio from the microphone and create a *free lossless audio coded*, referred to as FLAC, file and save it locally. This component of the module can be thought to act as the first stage of the process of collecting audio data and submitting it for transcription. It acts locally and directly accesses the computers microphone to record the data. The file consists of three functions, labelled record_audio, save_to_wav, and wav_to_flac.

2.3.1.1 Record Audio

In the record_audio function, a PyAudio stream is opened and input from the microphone is recorded in chunks of 1024 bytes at a rate of 16000 kHz for a duration of n seconds set by the parameter audio_length. The rate of audio capture at 16000 kHz is suggested by Google for their Cloud Speech-to-Tet platform, and 1024 bytes is the standard among audio applications.[25] Each chunk of audio is then appended to a python list and after the specified amount of time has elapsed, this function returns the python list consisting of the audio data.

2.3.1.2 Save to WAV and WAV to FLAC

The method used to convert the python list to the FLAC file required for submission to Google is divided into two separate functions. One function converts the list of audio data to a waveform audio file, referred to as a WAV file, and the other converts the WAV file to a FLAC file. The reason for this separation is that the WAV file is the originally recorded audio, without any form of compression, whereas the FLAC file is compressed using lossless compression. The reason for converting the WAV file is two-fold. One is that it reduces down the required storage space from multiple megabytes to the kilobyte range and the other reason is that the Google Cloud Speech-to-Text accepts FLAC files, allowing for quicker submission time, speeding up overall protocol generation.[25] The first function save_to_wav, takes in the list of audio data, the sample width, and the folder name for where the wav file should be saved. The second function, wav_to_flac, takes in the path to the WAV file and converts it to a FLAC file and removes the original WAV file to save space.

2.3.2 Voice Conversion

This file is responsible for submitting an audio file to the Google Cloud Speech-to-Text service and returns the transcription of the audio, along with a confidence score. While most of the intricacies of the machine learning model used to transcribe the audio are kept secret by Google, the non-proprietary and publicly available details behind Google's Cloud Speech-to-Text service are described in section 2.2.1. Google provides a prebuilt python package to interact with the Speech-to-Text API, allowing for easily transferrable and understandable code. The function named audio_submisssion_google performs the actual submission of an audio FLAC file to the Google Cloud Speech-to-Text service and receives the text back from the service. It reads in the audio file and stores it in a Google specific data type using the types.RecognitionAudio function.[25]

Once generated, the function then proceeds to configure the Speech-to-Text model, specifying the sample rate, file type, and the language, which is US English for this case. It then submits the configuration data and the modified audio file to the Google Cloud Speech-to-Text service, which then returns the results of the transcription: a string of the text along with a floating-point value for the confidence in the transcription, with 1.0 as perfect confidence.

2.3.3 Voice to Text

The final step in the process is located in Voice_To_Tex.py. In this file, the function audio_to_text_google acts as a wrapper for all of the code in the other two files, allowing for a single function call that allows for complete transcription of an audio sample. This function only requires two parameters. It requires audio_length, which is the length of time for the desired recording in seconds, and the second is a project name, with which it will save the FLAC file of the audio in a directory of the same name. As a whole, this function first creates an audio recording of the desired length using the built-in microphone, saves this audio sample as FLAC file, and then returns the text and confidence after submission to the Google Cloud Speech-to-Text service.

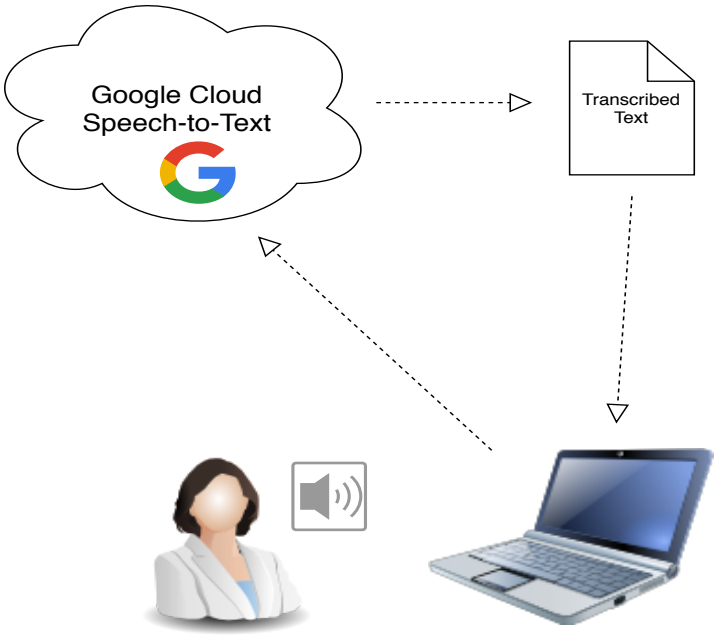

Figure 2. Schematic of Speech to Text System
The user records audio locally, which then gets sent to the Google Cloud Service, which then returns the transcribed text to the user.

Text Categorization

3.1 Purpose and Goal

After obtaining the text transcriptions from a given audio input, it becomes necessary to classify the instruction to direct further processing. This added step allows for optimal parsing of the instruction, as different instruction classes contain different information. Through analysis of the Opentrons library, it was determined that there are five distinct categories of instructions into which an instruction should be classified.[6] These five instruction classes are: initialization, transfer, distribute, wait, and mix. Initialization instructions correspond to the defining of container objects, such as pipette tips or a 96 well plate, and their location on the robot bed. The transfer and distribute classes are nearly identical, except that transfer steps require that the pipette obtains a new tip after moving a given volume liquid, while a distribute step uses the same pipette tip for multiple liquid transfers. The wait class pauses the machine for a given time interval, allowing experimental conditions to incubate. The last class, mix, modifies the previously entered transfer or distribute step, allowing the pipette to mix the liquid in the well after dispensing. With these five classifications, it becomes easier to design specific syntactic parsers to extract the necessary information.

3.2 Choosing a Classifier

There are many options when choosing a classifier for text, with varying degrees of mathematical and computational sophistication, with new methods being researched constantly, many with unique strengths.[26,27] The classifiers that were considered for use in this portion of the project were a Naïve Bayes classifier, a decision tree classifier, and a neural network. The theory behind each classifier was explored as detailed below, and a final decision was made selecting the Naïve Bayes classifier as the optimal choice.

3.2.1 Naïve Bayes

The Naïve Bayes classifier is a probabilistic classifier, and it functions by providing the classification that has the highest probability given the training data. The model takes in the training examples, extracts features from these examples, and then calculates the conditional probability that a given feature belongs to the corresponding class. The key assumption that the Naïve Bayes classifier makes is that the occurrence of each feature it measures is independent from any other feature. This assumption is normally false, especially with human language, however the models generally work extremely well in practice for text classification.[28,29] From all of these conditional probabilities, the model then classifies a new data point as the class that produces the maximum a posteriori probability. In the terms of text classification, the frequency of every word is tabulated and the conditional probability that a given word belongs to a certain class is determined. When given a new test example to classify, the model selects the class that is the most probable given the words in the phrase.

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k)$$

Figure 3. Naïve Bayes Model
The symbolic representation of the naïve Bayes model written as a function of classes and probabilities. It finds the class k such that the probability of the sentence belonging to the class is maximized.

3.2.2 Decision Tree

The purpose of a decision tree is to split the training data into two binary classes successively through the identification of features of the text that minimize a given loss measure.[30,31] The leaves of this binary tree are the desired classifications and thus the goal is to find the most accurate separations in each data partition to allow a particular piece of data to reach the correct leaf. The benefits of a decision tree come from the ease of interpretation and the intuitive concepts behind the classification. This characteristic is especially useful when using the model to make predictions such as disease classification, where the successive narrowing down of symptoms is similar to what would take place for a medical doctor.[30] As with any model, there are also a set of issues and complications associated with decision trees that limit their use. A relevant shortfall with using this type of model for the text classification is that there is a tendency of decision trees to overfit the data. This situation arises because the possible partitions for q data points into two partitions is $2^{q-1} - 1$.[30] Thus, the number of possible partitions grows exponentially with the amount of data available, which increases the chances of finding a particular model that classifies the data as desired. In theory this sounds beneficial, as there will be a model that works for the given data, however in practice, this means that the model has overfit the data, potentially to the point where any data point outside of the training space will be incorrectly classified.

In addition to the overfitting issue, there is also an issue with the stability of the trees. A decision tree propagates any error in the training data all the way down the tree, through all of its layers. Since the data was manually annotated, any misclassification or discrepancy in that training set will cause issues in generating an accurate model.[30] Human language offers a seemingly infinite number of ways to say the same thing, thus the high likelihood of overfitting and the extreme

propagation of error that is inherent in the decision tree model caused for the decision to remove the decision tree classifier from an option for the classification of instructions.

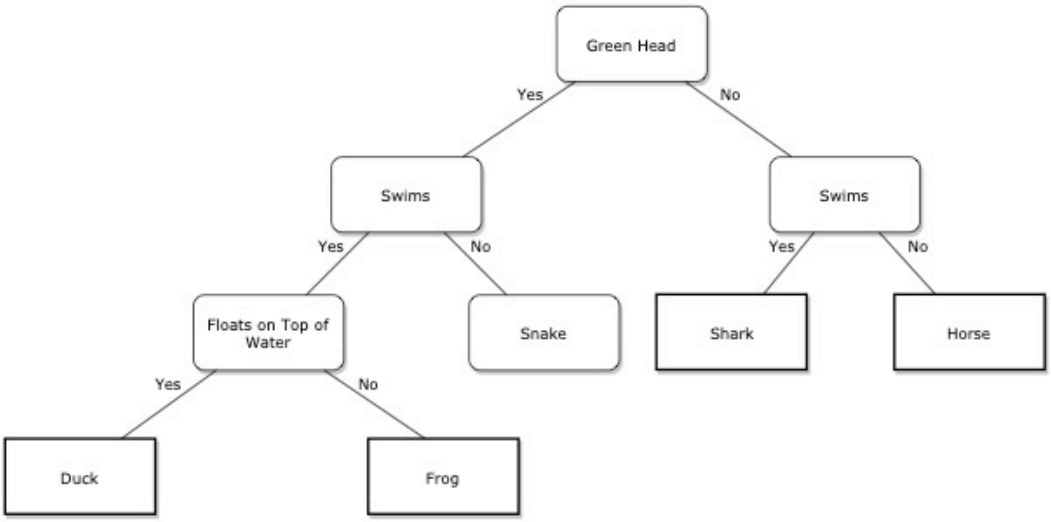

Figure 4. Decision Tree Schematic
An example of a decision tree with the leaves representing classifications of an animal with a green head to illustrate the concept.

3.2.3 Neural Networks

Neural networks are a nonlinear classification method that can be conceptually understood as representing the neuronal connections in the brain. There is a set of input nodes and output nodes, and layers of "hidden nodes" in between. In a fully connected network, each node is connected to all of the nodes in the next layer and their output, which can be the result of any function or computation the neurons perform on all of its inputs, to the next node is scaled by a scalar referred to as a weight. The goal of a neural network is to optimize all of the weights in the network such that given a piece of training data as input, the end result of the output layer will correctly classify the data point.[30] There are many benefits to neural networks, specifically relating to the massive increase in potential models that allowing nonlinearity provides. Furthermore, they can be extremely accurate and there are many different varieties of neural networks that allow them to better fit the specific task at hand.[32]

Although neural networks are becoming more well known for their versatility and power when creating predictive models for data, there are certain considerations that make their use impractical in a given situation. Similar to decision tree models, overfitting is a common problem with neural networks as there are thousands to millions of parameters to tune, namely the weights of the linkages between nodes. This large number of parameters means that over fitting to an insufficiently large dataset is highly likely. To combat this, weight decay methods have been developed to shrink weights towards zero as training progresses, eliminating weak connections and acting as a sort of pruning for the network.[30] Additionally, the cost of training the model and performing all of the necessary algorithms to correct the weights is normally prohibitive and impractical on personal computers. There are many other considerations, however the lack of size of the dataset generated, even after augmentation, on the instruction data makes the use of a neural

network impractical. There would be a high chance of severe overfitting, and there is a possibility that there is simply not enough data to allow the model to converge to a result that is useful for the purposes of this project. With these advantages and disadvantages compared between all three potential models, the Naïve Bayes Model was selected as the choice for this project. There are still other potential models that could be used for this task, however with the goal of a low weight, yet highly accurate model, in mind, a Naïve Bayes approach seems the most useful and suitable for this purpose.

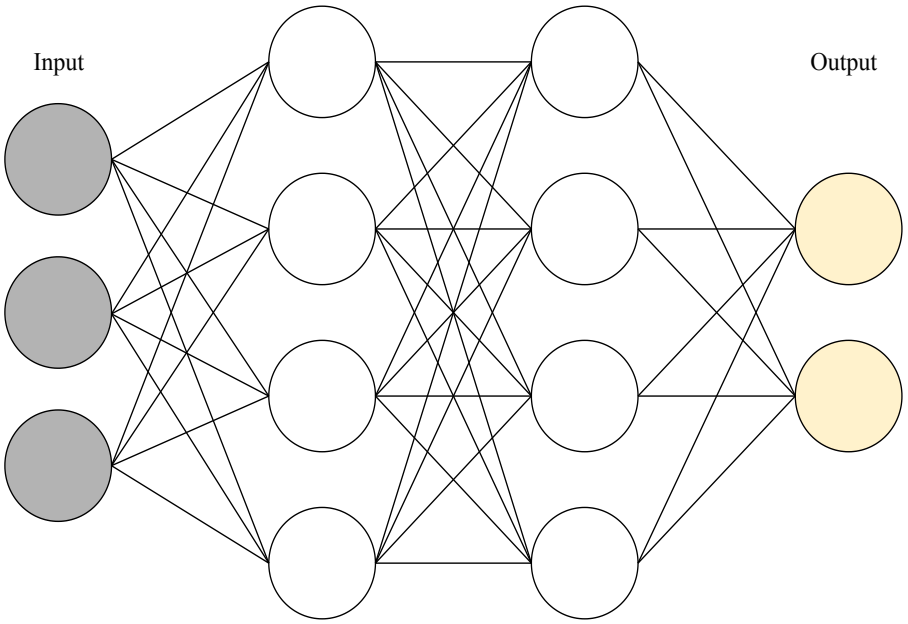

Figure 5. Fully Connected Neural Network
A representation of a 4-layer network with two hidden layers, 3 input nodes, and two output nodes. This network corresponds to a binary category classification of a three-dimensional input. Each line represents a connection between nodes, with information flowing to the right. These connections would be weighted and the combination of all the weights creates the network capable of classification.

3.3 Implementation of Naïve Bayes and Model Performance

As with any machine learning task, there needs to be training data and testing data to create and evaluate the performance of the model. The training data for the model had to be created specifically for this project, and to produce the most generalizable result, there needs to be a large amount of varied training data.

3.3.1 Training Data

To supply the necessary training data, a total of eight volunteers* from varying scientific backgrounds, ranging from no experience with scientific research to currently in science PhD programs, were recruited and each was provided with a list of four protocols and a set of instructions to write out the steps for each protocol as they would explain it to another person. Once this data was collected, each instruction was then manually annotated with its instruction class and then all individual instructions were combined into one csv file. A potential source of error to be aware of is that all participants were given the same four protocols, and as a result, the Naïve Bayes model will pick up on specific words used in the given protocols, despite the lack of any meaningful information relating to instruction class.[33] An example of this situation is the use of the locations of a container object, such as C1 in the given protocols. The model will then use the frequency of the appearance of C1 to help classify the statement as initialization. This problem had to be addressed, as the user can provide any location in the instruction and therefore the appearance of C1, or lack of, does not aid the classification.

To combat this issue, the data was augmented as follows. A list of the nouns, locations, and names of objects in all of the provided protocols was compiled. Additionally, a list of 5448 random nouns was created using an online random word generator. By iterating through each

* These volunteers were Tufts graduates from the class of 2018 and in alphabetical order by last name they were: Amy Austern, Sidney Beecy, Anjana Gupta, Connor Ko, Nick Metzger, Laura Quinto, Hannah Voelker, Lauren Witt

instruction, one of the words in the compiled list were changed to a random noun form the random noun list using the numpy.random.choice function.[34] A similar process was carried out for slot locations and for augmenting the volumes in the instructions, a random integer between 0 and 1000 was taken from a discrete uniform distribution using the NumPy numpy.random.randint function.[34] In addition to performing the augmentation, the number of training examples for the different classes were balanced, yielding a total of 200 examples per category. Since the Naïve Bayes computes probabilities, having an unbalanced training set will skew the predictions in the testing set to reflect the probabilities present in the training set. This augmentation process was performed 30 times, each with a random seed ensuring that different words would be substituted in for reach phrase. This was done because there was a chance although small, that word and number substitutions could be repeated, resulting in the same issues as before.

Once all of the training data was augmented, consisting of a total of 1000 training examples for each augmented data set, the data was then shuffled using the numpy.shuffle function, which shuffles the NumPy array in-place.[34] Once shuffled, an 80/20 split for the training set and test set were generated through random selection of phrases. The random seed was always 42 for shuffling and splititng, ensuring that the data was shuffled in the same way and that the same phrases would be selected for the training and testing set. This method prevents fitting the model to the testing set when selecting the best model, which would be the case if the best performing model was taken by doing random selections of the test and training set.

Data Augmentation Algorithm

1: **function** AUGMENT(*R*, *W*, *X, seed, S*)
2: **Input**: *R* (file path to csv containing all labelled data), *W* (text file with random words on each line for replacement), *X* (list of words that should be replaced), *seed* (Integer number for seeding random data augmentation), *S* (path to directory where augmented data should be saved).
3: **Output**: *D* (Path to csv file containing augmented data).

```
4:      Set Random Generator with seed
5:      Read in data from R into list
6:      Read in data from W into list
7:      class_counts ← dictionary to hold classification counts for each class
8:      a ← list to hold augmented sentences and classifications as 2-tuples
9:      for each sentence in R do
10:         sentence_class ←classification of sentence
11:         for each word in sentence do
12:             if word in X and class_counts[sentence_class] < threshold then
13:                 j ←random.choice(W)
14:                 a ← (sentence.replace(word, j), sentence_class)
15:                 update class counts for this sentence classification
16:                 break
17:         end for
18:     end for
19:     Save a as csv file at path D
20:     return D
21: end function
```

Figure 6. Data Augmentation Algorithm
Pseudocode for the augmentation of the training data. Each sentence is iterated through and a single word that is in both the sentence and the list *X* will be replaced by a randomly selected word from set *W*, keeping the sentence classification the same. This process is repeated until each category is balanced and it is then saved as a csv file for training.

3.3.2 Performance Evaluation

The Naïve Bayes model was implemented using the TextBlob python package, which has prebuilt classifiers that are already designed for the purposes of text classification.[35] Training a new model is as simple as calling a train method on the model object after specifying the location of the training data set, organized as a csv with the phrase and classification on each line. Each model was evaluated through the generation of a confusion matrix by using the results from predictions on the test set.[36] From this confusion matrix, the precision, recall, and F-score score can be calculated. The precision of the model refers to the proportion of true classifications over false classifications in that category. For example, if the model predicts 10 initialization steps in the testing set and of those 10, it got 8 correct, the precision would be 0.8. Therefore, precision acts a gauge for how confident one can be that the predicted classification is correct. The recall of the model refers to the proportion of predicted classifications for a given category to the total true classifications for a given class in that category. Continuing from the previous example, if the model predicts 10 initialization steps and there are actually 20 true initialization steps, the model would have a recall of 0.5.

		Actual Class				
Predicted Class		Initialize	Transfer	Distribute	Mix	Wait
	Initialize	19	0	0	0	0
	Transfer	1	18	1	0	0
	Distribute	0	2	19	0	0
	Mix	0	0	0	20	0
	Wait	0	0	0	0	20

Figure 7. Example of Confusion Matrix
Sample confusion matrix as would be produced by model evaluation. True positives (TP) are along main diagonal. False positives (FP) are column sum minus the true positive element. False negatives (FN) are row sums minus the true positive element. Precision for a category is defined as $\frac{TP}{TP+FP}$. Recall for a category is defined as $\frac{TP}{TP+FN}$. F-score is defined as $2 \times \frac{Precision \times Recall}{Precision+Recall}$

The F-score score is the harmonic mean of the precision and recall, allowing for an equal weighting of the precision and recall when evaluating the model.[32,36,37] For multiclass models, the score can either be micro or macro averaged. A micro averaged F-score score averages the false positives and false negative counts from all categories and then averages them together and calculates the F-score score. Conversely, a macro averaged F-score score computes the F-score score for each class and then averages the final scores together. Since all of the classes are balanced in the training data, a macro average is preferred as all classes are already contributing equally to the score.[36,37] The F-score score ranges from 0 to 1, with 0 being the worst case and 1 being perfect precision and recall. The top performing model was selected as the one to be used for text classification and it has a corresponding F-score score of 0.9867. This score was deemed satisfactory for the purposes of the project, as the majority of phrases will be categorized correctly and for the instruction that are not categorized correctly, it is most likely because the sentence structure is too challenging to parse and thus the user would have to reword the phrase anyway.

Table 2. Evaluation Metrics of Instruction Naïve Bayes Classifier
The precision, recall, and F scores following 10-fold cross validation of the augmented dataset. This F_{macro} score represents the macro average F-score, which is more suitable for the multiclass classification performed when compared with the micro average F-score.

	Precision	**Recall**	**F_{macro}**
Score	0.9853	0.9881	0.9867

Text to Code

4.1 Purpose and Goal

Converting instructions from natural language to Python code is the theoretical underpinning of the entire project, thus the methods through which this is accomplished is of paramount importance to accomplishing the mission of the project. The bulk of this portion of the project relies on tools from the machine learning field of NLP, where the goal is to understand the semantics of, and parse, human language for purposes of better human computer interaction and for automatic analysis of written documents. By developing a way to extract the pertinent information from the transcribed instruction from the user so that the necessary Python commands can be constructed, the project will be able to successfully go from human language to programmatic code. Due to the inherent creative nature of human language, it becomes challenging to develop a single model capable of extracting the necessary information from any sentence; so different methods are used for the different classified instructions, as instructions that convey similar meanings tend to have similar structures. Furthermore, having different methods of parsing leads to a cleaner overall program, as individual methods can be developed to suit the needs of a particular class of instruction.

4.2 Potential Methods

There have been numerous successful attempts to integrate human speech into the control of robots, both in the medical and industrial world.[14,19,38] The ingenuity in developing methods to extract information from speech is far reaching, with each method having its own strengths and weaknesses. Some methods rely on directly processing the speech itself and not the word content of the instruction, while others rely on the presence of specific words.

4.2.1 Strict Vocabulary

A frequent approach when training a robot to use voice commands as input is to have a set vocabulary, usually of two to three word phrases or single words, and to tie instructions to the utterance of these words.[14,19,39,40] This approach has the benefit of being able to link highly detailed commands for the robot to simple keywords. Since the vocabulary can be arbitrarily large, essentially every command of the robot can be tied to a word, with the challenge shifting from the robot, in its ability to understand, to the user, who has to remember the long list of commands to be able to do exactly what is required. This approach would certainly work for this project, however it is analogous to a brute force approach in computer science, where every possibility is tried when trying to find the correct answer. It is guaranteed to work, but the effort and resources needed to perform that task may be impractical.

4.2.2 Duration and Pitch

Extensions of the strict vocabulary method that allow for a greater depth of command in a more intuitive manner rely on the pitch and duration of particular phenomes in the instructional phrase. When commanding a robot to move a particular distance, the “move” command will most likely be set to a predetermined distance. Thus, when trying to move for distances greater than that set length, the instruction may have to be spoken more than one time. To combat this, some

researchers have developed methods using Hidden Markov Models to take in the length of the word into account when deciding how far to move.[38] For example, the command "move left" may move 2 centimeters to the left, while saying "move leeeft" may move it 4 centimeters. This lengthening of the phrase to convey more meaning feels natural and is thus a good approach that does not require further extension of the command library.[38] Programmatically, this does add a bit of complexity, but since the goal is to have an easy to use and enjoyable user experience, it is certainly within reason.

Similar in concept to the lengthening of a phenome, another way to convey added information in the same word is to alter vowel and pitch. Voice control has important benefits in making technology more accessible to individuals with disabilities that may prevent the refined motor coordination for activities such as typing or moving a mouse. One such method for utilizing the vowel sound of words is in a process called vowel mapping, which ties specific vowels to directions, such as the vowel a in "cat" meaning move the mouse up.[15] This mapping can also be extended to include pitch, where varying pitch in a specific direction would result in a different or more exaggerated command for the robot. There are demonstrated results of people being able to use these techniques to control complex robotic arms, with multiple joints and angles that need to be controlled for and thus this is certainly a feasible method for enhancing the voice control methods to allow for an even greater capacity of control.

4.2.3 Semantic Parsing

An entirely different approach which separates itself from the strict vocabulary and speech attribute methods is seeking to understand the deeper meaning of the language through semantic parsing. In this method, a language parser model is developed that is designed to break down a given sentence into all of its corresponding linguistic parts that can be used to extract meaning.[12] In an innovative approach to this problem, the Natural Language Processing Group at Stanford University have developed a publicly available semantic parser named SEMPRE.[12] When provided with a statement, this parser is able to break down the sentence into "logical forms" which explicitly describe the content of the statement in a way that a database can be queried and the answer retrieved. By using this parser, it becomes possible to break down an instruction into a logical form that can be easily parsed and developed into code.

4.3 Method Selection

A functional method to parse and generate code from verbal instruction requires a combination of many different methods as each method individually is not capable of performing all of the required tasks for this project. More macroscopically, since the accuracy of an experiment or scientific study relies on the performance of this program, a range of different methods had to be applied to generate the overall best performance. The portion of the project discussed in **Chapter 3**, the development of the text classifier, allows for this compartmentalized approach as at least five different methods may be applied, one to each instruction category. Since instruction statements that belong in the same instruction class tend to have similar structures, a particular method that can take advantage of that structure can be developed and utilized.

Overall, all of the developed methods do utilize a variation of the strict vocabulary method and semantic parsing, where specific phrases classified and searched for in the instruction. For this

project, SEMPRE was not utilized as the necessary training data could not be generated in sufficient quantity to get the most out of the tool.[12] Additionally, a grammar structure would have to be developed to guide the model on how to pick out features for each specific class. This would require that one or more grammars would have to be identified and coded for each parser, which soon becomes unwieldly with testing and generalizability.

Due to the complex grammatical structure of some sentences, a simple strict vocabulary approach is not applicable by itself. To aid this approach, a part-of-speech tagger was implemented to label every word in the sentence with its part of speech. The Python Natural Language Learning Toolkit library, which has part-of-speech taggers trained on massive text corpuses that are capable of tagging virtually any sentence. These tags are useful in that the objects of the sentence and their relation to verbs will help provide the link to a resource in the experiment, making it easier to extract the information from the spoken instruction.

4.4 Code Organization

All of the code, supporting files for data set generation, and the exact implementation of the concepts discussed in this section, and all previous sections, can be seen in the documentation and the source code provided at:

https://github.com/KevKap/Opentrons-Voice-Control

With all of these parsers, many common lab instructions that are provided in English can be translated to Python code, thus allowing for complete voice control of protocol development for the Opentrons platform.

4.4.1 Initialization Parser

4.4.1.1 Non-Pipette Objects

The initialize_parser function is responsible for extracting the necessary information to initialize container objects and pipettes for use during the experiment. The Opentrons library has a set number of pre-built containers, which are lab items such as 96 well plates or 2 milliliter tube holders, and thus it became necessary to match the container stated in the instruction to the same name as a container in the library. For this task, another Naïve Bayes classifier was constructed, but with training data that corresponded to classification as a specific container object.[35] The dataset was constructed from the training data provided by the volunteers by manually extracting the initialize commands and substituting different containers into the statements to generate training data for all of the frequently used containers. Once the initialization command is classified

Table 3. Evaluation Metrics of Container Naïve Bayes Classifier
The precision, recall, and F scores following 5-fold cross validation of the augmented dataset. This F_{micro} score represents the micro average F-score, which is more suitable for the multiclass classification performed when there is a class imbalance. All three values are the same due to equal likelihood of a false positive as a false negative.

	Precision	**Recall**	**F_{micro}**
Score	0.9637	0.9637	0.9637

as corresponding to a specific container, the location, or slot on the robot bed, is extracted using a regular expression, or regex, search for a letter immediately followed by a digit. If no location of this form can be found, the phrase “No slot location for (container object) detected. Please replace this text with slot ID in quotes (e.g. “A1”)” in the code where the slot location should go.

Once the container and location information are obtained, the name of the container, such as the user calling a 15-milliliter conical tube “water”, has to be extracted, as the user will refer to this initialized container with this name. The name of the container is taken as the last word of the instruction. Although there are ways to name the container without having the last word of the instruction be the name, such a significant portion of the data gathered from the volunteers indicated that the container name is normally present at the end. This hardcoding in of a solution does have the potential to cause issues with the user experience, but this issue among others is further discussed later on. The only exception to this rule is with the trash and tiprack object. When an instruction is classified as the trash container, the name is automatically adjusted to be trash, and a tiprack is named with its classification, with a substitution of the hyphen for an underscore such as “tiprack_p100”. The user will never refer to this object outside of initialization, so the consistent naming will be important for initializing pipettes. With all three pieces of information, the Python command for initializing a container can be completed.

```
plate = containers.load('96-flat', 'A1')
```

Figure 8. Example of Container Initialization Code

Initialization Parser Algorithm

```
1: function INITIALIZE_PARSER(I)
2: Input: I (transcribed instruction provided as string).
3: Output: S (string of Python code corresponding to instruction I).

4:      container ← container classification (provided by container Naïve Bayes model)
5:      if container not equal to pipette then
            6:      slot_location ← regex search of instruction for alphabet character
            followed by a digit
            7:      if container not in [tipracks, trash] then
            8:          name ← I[end]
            9:      else if container equal to trash then
            10:         name ← trash
            11:     else if container equal to tipracks then
            12:         name ← container
            13:     return S(name, container, slot_location) # This is formatted as described
            in the sample initialization command for a container object
            14:     else then
            15:     pipette_type ← regex search of instruction for lowercase p followed by
            any number of digits
            16:     location ← regex search of instruction for the word pipette followed by
            upper or lowercase a or b
            17:     tiprack ← digits from pipette_type
            18:     return S(pipette_type, location, tiprack) # This is formatted as described
            in the sample initialization command for a pipette object
            19: end function
```

Figure 9. Initialization Parser Algorithm
Pseudocode for the initialization parser. An instruction is first classified by the container Naïve Bayes classifier and sorts by a pipette or non-pipette object. A non-pipette object is further classified into a trash or tiprack container. A trash object is given the name trash and a tiprack container is given the name of the tiprack, whereas a different object is taken as the last term in the instruction. Initialization information for pipette objects are extracted via regular expressions.

4.4.1.2 Pipettes

If an initialization instruction is classified as a pipette, a slightly modified version of parser is implemented. Since pipettes are referred to by their volume, such as “p1000” or “p200”, a regex search is preformed looking for a lowercase p immediately followed by a potential space and then any number of digits. This extracts the type of pipette and to determine if the pipette is placed in pipette slot A or B, another regex search is performed looking for “pipette” followed by a space followed by a capital or lower-case A or B. The motivation behind this seemingly odd search is that the pipette slot is referred to as “pipette a” or “pipette b” thus searching for “pipette” followed by a space then either A or B allows for the correct slot to be initialized. If no pipette volume or slot can be found, the user has to input the correct data into the code by replacing the places in code with the following phrases with the respective information:

- “No pipette type detected. Please replace this text with pipette type (p10, p100, p200, p100) in quotes. Also adjust the max volume (no quotes) to volume in microliters and the tiprack to tiprack-max volume ul e.g. tiprack-200ul”
- “No pipette channel detected. Please replace this text with pipette channel in quotes ("a" or "b")

Once this information is provided, the volume of the pipette is added into the class attribute pipette_list, which holds the volumes of the available pipettes. With all of this information, the pipettes can be initialized and are capable of being used in the transfer and distribute steps.

```
p100 = instruments.Pipette(axis="b", max_volume=100, tip_racks=[tiprack_100])
```

Figure 10. Example of Pipette Initialization Code

4.4.2 Transfer and Distribute Parser

The transfer and distribute commands for the robot are virtually identical, so the parsers for the two commands are the same and distribute_parser simply calls transfer_parser and replaces the word transfer with distribute in the code. The two commands are distinguished by how the robot handles the tips after moving the liquid. In a transfer step, the specified volume is moved to the designated wells and the tip is placed in trash. In a distribute step, multiple transfer steps may be combined without changing the tip in between. When to use each one is more of a matter of preference, however for larger sample transfers of the same resource, the Naïve Bayes developed in **Chapter 3** will select the distribute classification.

This parser takes advantage of the part of speech tagging function from Python's Natural Language Processing Toolkit library to determine the correct source and destination in the transfer step.[13] Although in spoken language, the phrase "move x into y" and "move x to y" mean the same thing in this context, the part of speech tagger tags them as different parts of speech, as "TO" is its own category. Thus, the first step is to replace any occurrences of the word "into" with the word "to." Once this is complete, the instruction is tagged with its parts of speech. The original instruction, tagged instruction, a starting part of speech, and an ending part of speech are then passed into a function that extracts the reagent name and location for the source reagent, which corresponds to a start and end part of speech of "IN" and "TO" respectively. This data extraction is performed in multiple steps and functions, with each function extracting a specific piece of data. First, the reagent, or container, name is extracted by identifying the noun in the sentence fragment, provided it is not column, row, or well. Second, the sentence fragment is classified into either specifying the location via columns, rows, wells, or none of these. For a column location, the ordinal or cardinal value provided is mapped to the corresponding English letter and returned back

as the column ID. For a row location, the ordinal value has to be converted into a cardinal value, which is then returned back as the row ID. The well ID extraction is easier, as it is always the word following the word “well” in the instruction, so a separate function returns this well value if a sentence fragment is classified as containing a well ID.

Following the ID extraction, a dictionary with “c”, “r”, and “w”, representing column, row, and well, respectively, for keys, and Python f-strings as values representing the properly formatted Python command for that given location and reagent name. This formatted string is then returned and saved as the source parameter

The volume of liquid to transfer is selected by calling the volume_select function with the tagged statement as a parameter. This function locates the number in the instruction, using the “CD” tag, or “cardinal digit”, which corresponds to the volume, and it also locates the volume measurement, such as milliliter or microliter, which would be tagged as “noun plural,” “noun singular,” or “adjective,” and adjusts the volume number accordingly before returning the value since pipette transfer volumes are interpreted to be in microliters in the Opentrons library. Once the volume is extracted, the last selection that needs to be made is which pipette to choose. To have the highest accuracy, the pipette chosen should be the one with the maximum volume that represents the lowest upper bound to the volume needing to be transferred. In an example scenario, if a p200 and a p1000 are installed, and the user wants to transfer 205 microliters, the p1000 is going to be selected.

Following the volume extraction, the same method that was performed to extract the source information is then performed on the instruction fragment between the “TO” and “.” parts of speech, to extract the destination information. This information is saved in the destination variable and with all of this information, the transfer, and subsequently the distribute, since the distribute

function just calls the transfer parser and replaces “transfer” with “distribute”, Python commands can be completed and returned.

```
p100.transfer(75, trough, plate.cols('A'))
```

Figure 11. Example of Transfer Code

Transfer and Distribute Parser Algorithm

1: **function** TRANSFER_PARSER(*I*)
2: **Input**: *I* (transcribed instruction provided as string).
3: **Output**: *S* (string of Python code corresponding to instruction *I*).

4: *tagged* ← part of speech tagged instruction
5: *source* ← words between “preposition or subordinating conjunction” tag and the word “to” and looking for columns/rows/wells and creating the appropriately formatted source
6: *volume* ← “CD” (cardinal digit) tagged word that is converted to microliters based on adjective or noun tagged words # This looks for milliliter or microliter in adjectives or nouns and multiplies the volume by 1000 if milliliter is found
7: *destination* ← words between “to” and end of sentence (‘.’ tag) and looking for columns/rows/wells and creating the appropriately formatted source
8: *pipette* ← pipette based on pipette with max volume as least upper bound of *volume*
9: **return** *S(pipette, volume, source, destination)* #This is formatted as described in

Figure 12. Transfer and Distribute Parser Algorithm
Pseudocode for the parsing of transfer and distribute statements. An instruction statement is supplied and each word is tagged with its part of speech. The sentence fragment between the “IN” and “TO” tag is extracted as representing the source and the reagent name and location information if provided is extracted. The volume is then extracted as the cardinal digit and converted to microliters. The sentence fragment between the “TO” tag and the end of the sentence represents the destination and the reagent name and location information if provided is extracted. Finally, a pipette is chosen based on the least upper bound of the liquid volume being trasnfered.

4.4.3 Mix Parser

The mix parser is unique in that it edits the previously parsed instruction and does not add in its only line of code directly. To mix a well after transferring a liquid, a mix_after parameter has to be added to the transfer call. The previous step from the protocol is retrieved and the number of times to mix is taken from it. If no set number of mixes is provided, a default of two times is returned. Additionally, the volume of the previous transfer is halved, and this half volume is the amount used to mix the well with. Once these two parameters are obtained, they are substituted into the previous command in the proper location and thus the mix command is now implemented into the code.

```
p100.transfer(75, trough, plate.cols('A'))
p100.transfer(75, trough, plate.cols('A'), mix_after=(2,130)
```

Figure 13. Example of Transfer Code with and without Mixing

Mix Parser Algorithm

```
1: function MIX_PARSER(I)
2: Input: I (transcribed instruction provided as string).
3: Output: S (string of Python code corresponding to instruction I).

4:      n ← 2
5:      if previous step is transfer step:
6:          tagged ← part of speech tagged instruction
7:          for each word in tagged do
8:              if word tag == "CD" then
9:                  n ← word
10:                 break
11:         end ford
12:
13:     mix_volume ← 0.5* Extracted volume from previous transfer step
        14:     return S(n, mix_volume) # This is formatted as described in the sample transfer
        function with mixing
            15: end function
```

Figure 14. Mix Parser Algorithm
Pseudocode for the parsing of a mix statement. The words in the instruction are tagged with their parts of speech and the tagged instruction is then iterated through and the tag corresponding to a cardinal digit, "CD", is then taken as the number of times to mix. Otherwise the mix number is set to two times by default. The volume selected is half the volume of the transfer, which ensures no air bubbles will be created.

4.4.4 Wait Parser

The wait parser is used to add a delay to robot. This allows for samples to incubate for a specified amount of time or delay any step at the user's discretion. This parser utilizes the part-of-speech-tagger to locate the cardinal digits as in the transfer parser and finds the corresponding time measurement, either minutes or seconds. The function time_list_generator is called with the tagged statement, and the corresponding minutes and seconds are extracted using a stack structure. These values are returned as a list of lists, with each sub-list being of the form [time measurement, number]. Back in the wait_parser, each sub-list is joined with the "=" delimiter, as required by the Opentrons delay function syntax, and then each for the new strings of "time measurement=value"

are joined with a comma separating them. A random pipette is then picked to act as the delay and the proper delay syntax is then filled in with the appropriate amount of time.

```
P100.delay(minutes=20, seconds=30)
```

Figure 15. Example of Wait Code

Wait Parser Algorithm

```
1: function WAIT_PARSER(I)
2: Input: I (transcribed instruction provided as string).
3: Output: S (string of Python code corresponding to instruction I).

4:      tagged ← part of speech tagged instruction
5:      stack ←  initialize empty stack
6:      for each word in tagged do
7:          if word tag == 'CD' then
8:              stack.push(word)
9:          else if word tag in ['NNS', 'NN', 'JJ'] then
10:             if min in word then
11:                 minutes ← ['minutes', stack.pop()]
                else if sec in word then
12:                 seconds ← ['seconds', stack.pop()]
        13:     return S(minutes, seconds) # This is formatted as described in the sample wait
        function.  # The pipette for waiting is just chosen at random.
            14: end function
```

Figure 16. Wait Parser Algorithm
Pseudocode for the parsing of a wait statement. The words in the instruction are tagged with their parts of speech and the tagged instruction is then iterated through and the tag corresponding to a cardinal digit, "CD", is then taken as the number of minutes or seconds is put on a stack. Otherwise, if the word is a noun or an adjective, then if "min" is in the word, the number is removed from the stack and paired with minutes. The same process is done if the word is seconds.

Guided User Interface

5.1 Purpose and Goal

A guided user interface is designed to allow the user to interact with the program in an intuitive and productive manner, without having to see all of the intricacies of the program. In line with the purpose of this project, the design and features of the GUI were developed to be straightforward, encourage users to develop more complex protocols, and reduce the overall learning curve associated with operating the Opentrons robot. For the user, the GUI is the only interaction she has with the program and the voice to code platform, so it has to utilize all of the features and tools developed in a way that makes sense without the user having to understand the actual mechanics of how the project implementation works.

5.2 Features

At the start of the program, it asks for the user input for the Project name. This name will be used as the name of the protocol, along with the timestamp of when the protocol was created, allowing the user to have many versions of the same protocol, with the option to rename the file using the system's file explorer always remaining should the name need to be changed later on. The logistics of the implementation of all of these features is found in section **5.3 Code Organization**.

5.2.1 Recording the Instructions

Clicking the "Record" button in the center of the interface initiates an audio recording for five seconds. Once clicked and the screen displays "Recording…" the user can begin saying the instruction and once the five seconds are up, the audio file is submitted to the Google Cloud Speech-to-Text API and the written text is returned and presented on the screen. If the written transcription is correct, the user can simply click the next arrow located on the bottom right of the

screen to begin recording the next instruction. To re-record over an incorrect instruction, clicking the “Record” button while the earlier recorded instruction is displayed will overwrite that text. This feature allows the user to edit the protocol in any place, at any point, maximizing control and customizability.

5.2.2 Step Navigation

Since a protocol is constructed stepwise, with each instruction building from the previous, the navigation to switch between steps was designed to reflect this. At the bottom left and right of the program window, there is a left facing arrow and a right facing arrow, which represent the previous step and the next step respectively. Clicking the forward button moves onto the next protocol step, and if it is not previously recorded, it tells the user to click the “Record” button to record the next step of the protocol. It does not let the user proceed to the following step unless something is recorded, so there cannot be any gaps in the protocol production. Similarly, the user cannot move to an earlier step than the first step. Further navigation is provided by the “Insert Step” and “Delete Step” buttons. When pressed, the “Insert Step” button inserts a step in the current position, moving the currently displayed step to the next position and keeping the previous instruction as previous. The “Delete Step” button deletes the current step being shown and joins the previous and next step together and shifts the display to the next step. Both of these methods are implemented as simple double linked list methods with edge case handling for the first position and last position, as well as the first position when no other steps are present for the “Delete Step” method. These features allow for significant control of the protocol during editing and enables the user to go back and edit the protocol to be exactly as intended, without the need to re-record anything.

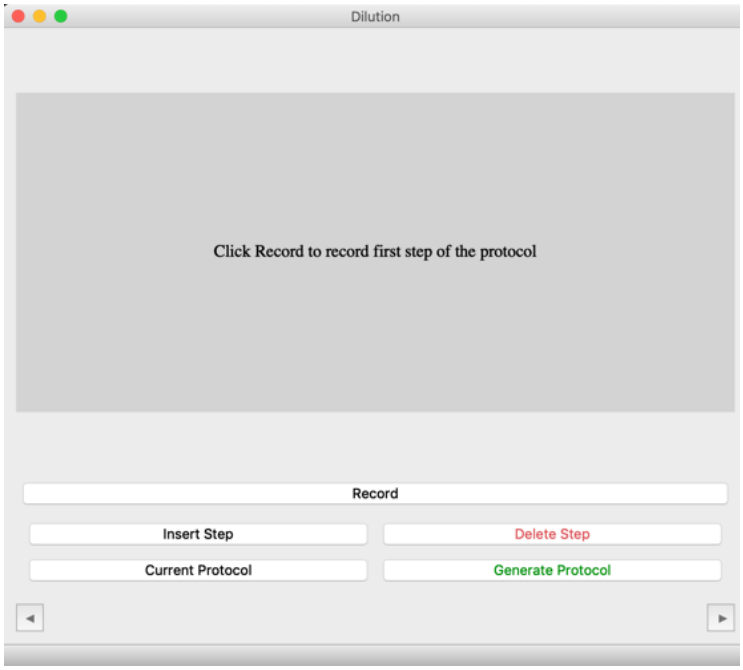

Figure 17. GUI Window for User Interaction

The user can generate her protocol using the GUI interface as illustrated. The Record button allows for recording of a new instruction, the Current Protocol button allows for all previous steps to be displayed, and the Generate Protocol button creates the Python protocol from the recorded instructions.

5.2.3 Error Handling and User Confirmation

The error handling in the GUI relies on the user to check to make sure the written transcription of the instruction is correct, by advancing to the next instruction. By advancing to the next instruction, the previous instruction is taken as correct and once clicked, the "Generate Protocol" button will use the written instructions that have been saved up to that point. The user, at any point, may click the "Current Protocol" button, which will display all of the steps of the protocol that have been generated. This serves as a sanity check for the user to make sure she is on track and that there are no missed steps. Once the user is ready to generate a protocol, she will click the "Generate Protocol" button, which will then ask the user if she wishes to generate the complete protocol. This check just makes sure that the user did not click the button by mistake and

if she did, clicking “No” returns back to the general recording window. Once “Yes” is clicked, a window displaying all of the transcriptions in order is displayed, allowing the user to read through all of the spoken instructions to confirm that all instructions are present. Clicking “No” returns back to the main recording window, as would be expected.

After gathering UI feedback from an active biomedical researcher, an additional confirmation step was implemented where a summary of the spoken steps is provided.[41] This summary provides the translation of the spoken instruction into a standardized format for a given instruction category, highlighting the extracted information. These summaries allow the user to judge if the Python commands will be correct, acting as a pseudocode for the user to judge. This extra step allows for the peace of mind that the robot will perform the intended actions. Furthermore, if the instruction is not correct, it allows the user to edit the protocol before a file is saved, allowing for a re-recording of the instruction with a different phrasing, reducing the total time to debug the protocol and also lending more confidence to its capabilities. By clicking “Yes” the program will then autogenerate the Python protocol and save it to the desired location which is selected via a typical file navigation window. Clicking “No” will bring the user back to the general recording window, allowing for the rephrasing mentioned earlier.

Instruction Category	**Summary Skeleton**
Initialization	Setting ***container_name***, a ***opentrons_container_name***, in location ***slot***.
Transfer/Distribute	Moving ***volume*** microliters from ***source*** to ***destination***.
Mix	Moving ***volume*** microliters from ***source*** to ***destination*** and mixing ***X*** times.
Delay	Pausing for ***X*** minute(s) and ***Y*** second(s).

Figure 18. Instruction Summary Skeletons
These skeletons will be populated with the corresponding extracted information from the spoken user instruction. A list of each of these summaries will be provided to the user as a final check before generation of the Python protocol.

Further error handling has also been implemented, consisting of an error readout to the top of the Python file illustrating the offending instruction and instructing the user to go to the specific line in the code which could not be generated. This instruction is in the location where the specific data which could not be parsed should go and it provides detailed information on how to replace it with the desired phrase, including examples. This error handling should not come up often, as the user can see in the summary step if the proper information was extracted, however it does act as a final fail safe for the user to make sure there are no large errors that could affect the experiment.

A further enhancement that was suggested was placing the verbal instruction above the corresponding Python command as a comment.[41] This addition was successfully implemented and allows for greater confidence and understanding of the code when reading through the file. Furthermore, it also acts as a record of what was spoken and the exact way the protocol was generated, which aids reproducibility and will help the same user understand what is going on when rereading the protocol at a later date, since all of the steps will be summarized in common language. These additional error handling methods and UI enhancements reduce the chance for confusion or misinterpretation for a given step, and also give the user greater confidence in the implementation and creation of complex protocols.

5.3 Code Organization

The GUI was created using the PyQt5 library.[42] This library constructs the GUI through the use of several class objects, primarily the QMainWindow, QWidget, and QThread classes. In the implementation of this project, four class objects were used and combined to allow for the fully functional GUI.

5.3.1 Instruction Node

To save the instruction transcriptions as they are provided, a simple doubly linked list was implemented. In addition to an instruction field and previous and next fields, this structure also has a data field for the confidence and for the step number. These three data fields allow for the complete storage of all components of the instruction. The average of all the confidence serves to provide a rough estimate on the confidence in the Python protocol. To prevent the GUI from allowing the user to advance to steps beyond what is recorded, it will not move on to the next step if the instruction field is not provided on the tail node. When the advance button is clicked, a new instruction node is generated which prompts the user to record a new instruction. To solve the reverse problem, the back button will only move to the previous step if the step number is equal to one or greater. Movement between pre-existing steps equates to traversing the linked list as would be expected.

5.3.2 Main Class

The main window of the GUI inherits from the QMainWindow class, which provides all of the necessary functionality to open up a blank program window to allow for the placement of the QWidget class. In the class *Main*, the functionality in the top menu bar is set up and the dimensions of the larger window are set. In the top menu bar, the user is able to cancel the current protocol and start a new one and generate the currently made protocol. Furthermore, the *Main* class

allows for the naming of the protocol in the initial dialogue box that pops up. The content of the main window is developed in the *VoiceControl* class, while the *Main* class acts as a container to hold the main GUI functionality code and all of the features and capabilities the user interacts with.

5.3.3 VoiceControl Class

The *VoiceControl* class inherits from the QWidget class and this is the actual object that the user interacts with during the recording of a protocol. In addition to the separation between the main window and the widget class, the *VoiceControl* class is broken into a "User Interface" section and a "Button Clicks" section. In the "User Interface" section, the dimensions of the inner window are set, the geometric layout of the buttons are defined, and the QThread for running the speech to text transcription code is set up. Additionally, the function to allow for the text display that allows for the recorded instruction for the current step to be displayed to the user is defined. This section of the *VoiceControl* class provides all of the visual and interactive components of the project, while the "Button Clicks" section is where the functionality associated with the project is implemented.

The "Button Clicks" section consists of seven functions, each of which is tied to the press of a corresponding button in the GUI. Out of these seven functions, two are related to the actual speech to text conversion. The function record_button_click initializes a QThread, using the *Transcription* class, and to this thread, attaches the transcription_processing function which appends the results of the audio_to_text_google function described in **Chapter 2** to the instruction list and displays the results of the transcription to the screen. Error handling is performed by checking to see if any results came back from the audio submission and if the result is None, the program alerts the user "No audio detected. Please try again." This error handling prevents a blank instruction from being appended to the instruction list, and thus prevents an edge case when performing the text to code conversion.

The next two functions correspond to navigation through the instruction list, allowing for the user to go back and edit previously entered instructions or to advance to the next instruction to add a step to the protocol. The advance function is a slightly more complex version of traversing forward in a doubly linked list. It first checks to see if it is at the tail and if it is, it creates a new tail, updates the current instruction step, links the previous *InstructionNode*, and instructs the user to "Press Record to record the next step of the protocol." If it is not currently at the tail, it will simply navigate forward and either display the instruction if it was previously recorded or ask the user to record the instruction. The latter case is when the user advances when currently at the last step of the protocol and then moves back to check another step before making a recording. This is an important edge case to handle as the user may need to refer to what was previously stated when making a new step, causing this situation to happen frequently.

The last components of the GUI are the three functionalities for generating the python protocols given the written instructions. The current_protocol function is tied to the "Current Protocol" button and this function iterates through the Instruction List backwards from the current step and appends each instruction to a normal python list. It then reverses the list to generate the steps in the proper order and joins the list with a newline separator between them and displays this generated string to the console, providing the user with the current protocol up to the current step. This functionality allows the user to check progress and catch any mistakes in real time as the protocol is generated. The generate_protocol function is connected to the "Generate Protocol" button and this function opens up the necessary tree of dialogue windows to have the user confirm that she wants to generate the protocol into Python code after checking it over again and then calls the compile_instructions function accordingly to generate the code. Once the file is successfully generated, the GUI then prompts the user to select if she wants to generate a new python protocol

or exit, allowing the whole procedure to either restart or close out. The compile_instructions function is what actually generates the Python code from the given instruction list, and this is performed by instantiating the *CodeGenerator* class and saving the results from the complete_generator method in *CodeGenerator* in a Python file. In addition to the actual python commands, this function writes the average confidence of all speech transcriptions to the bottom of the file as a comment and also names the file using the protocol title the user enters at the start of the program and the current date and time in Year-Month-Day-Hours-Minutes-Seconds format. It provides a file navigation GUI for allowing the user to select where to save the file.

5.3.4 Transcription (QThread) Class

This class provides the method necessary to submit the audio file to the Google Cloud Speech-to-Text service in a separate thread. This submission has to be run in a separate thread because the action of pausing to record the instruction causes PyQt to pause, preventing the display window from updating and therefore preventing the alert to the user that it is okay to record. By running a separate thread with this functionality, it allows the GUI to function as expected and alerts the user when to begin speaking and returns the transcription of the audio once and displays it to the screen upon cessation of the recording.

Discussion and Limitations

The development of a voice control feature for the Opentrons robotics platform required tools from varying areas of computer science, including probabilistic machine learning models, natural language processing, and GUI development. The combination of these different fields resulted in a user friendly and successful application that was able to take in everyday spoken English and create a Python program capable of running an assay on the Opentrons robotics platform. Although focused on this particular use case of the Opentrons platform, the generation of the protocol serves as the applied example of a broader development towards moving coding into an easier and more approachable field.

Through the classification of verbal instructions into categories, optimized processing techniques could be applied to extract the desired information. Although this method results in a less generalizable speech processing system, the addition of new categories is fairly simple through extension of the Naïve Bayes model and thus new categories and processing techniques can be added as needed, without potential interaction with the pre-existing categories. The ease of expansion that comes along with this method counters the lack of generalizability in each processing situation. The application can be extended to include any category and as, in this case, Opentrons, adds more features to the robot, the code can be easily extended to meet these new features. Through this modular approach to the problem, fixing errors become easier, redundancy is removed, and expansion is painless. As will be explored further, the Naïve Bayes model approach also adds rigidity that prevents immediate extension to different platforms, however conceptually, with the correct training data, the skeleton of the process would work with any code generation procedure. With this approach, it is also important to discuss the limitations that hinder its use and detract from a completely intuitive and hands-off approach for the user.

6.1 Residual Reliability on User

The most obvious limitation of this project is the focus on an English-speaking user. Although English is extremely wide spoken, lack of fluency in the language may make this application not usable, depending on the phrasing of the instructions. This language requirement, although not too limiting, does immediately restrict the use of this application to the English-speaking population. Despite this drawback, the Google Cloud Speech-to-Text service does allow for transcription into multiple languages, thereby only required new language specific processing techniques if the model were to be extended. Since there is nothing unique about English in the scope of this project, the methods developed are language agnostic, allowing for the incorporation of additional languages in the future.

In addition to being English only, some instruction categories require a more rigid sentence structure, although the processing techniques implemented are a lot more flexible and allow for more natural speech than previous methods [14,15,19,39] For example, the transfer parser, which creates the Python command for a transfer instruction, requires that the destination of the transfer be at the end of the sentence. An example of this structure is "Transfer 15 microliters of DBPS to cells," where the location of DBPS and cells are defined by an initialization instruction. This sentence structure is natural and represents the structure of the significant majority of transfer steps in the training data. A conceivable alteration of this structure would be the sentence "Transfer 15 microlites to cells from DPBS," although this structure feels slightly unnatural, it is possible that a user may provide this instruction as a result of a lack of fluency. The current application does not have a way of handling this deviation of expected structure. This lack of acceptance of all possible sentence structures may require the user to rerecord the instruction or protocol multiple

times, however this situation should be in the minority of use cases for fluent or near fluent English speakers.

Beyond the English-centric language component of the project, the user also has to be able to manually set up the robot and check over the produced protocol. This does not require a complete understanding of Python code to do, as comments in the code will guide the user on what to correct if it was unable to extract a particular piece of information. Being able to set up the robot is an inherent limitation in the project, as the user cannot just take the robot out of the box and connect it to this program and begin creating protocols. There is moderate setup required once a protocol is created, including initializing the exact locations of all containers and pipettes through the official Opentrons app. Once initialized, unless physically changed in a new protocol, these configurations are saved so running the same experiment two or more times a row does not require re-configuration of the machine and the containers. Since this limitation cannot be avoided, the Opentrons app was designed to make this process intuitive and thus it should not prevent a large barrier to its use.

6.2 Naïve Bayes Model

The Naïve Bayes model is probabilistic, meaning that it selects the most probable category given all of the training data by evaluating the probability of the instruction having a given word and belonging to a specific category. Therefore, the accuracy of the model architecture heavily relies on the training data to provide a representative sample of the actual data the model should be categorizing. Given this reliance on the training data, if the training data has an abnormal frequency of a given word in a specific category, it will identify this word and use it as a strong basis for category selection. This issue should be mitigated through the augmentation and randomization of the data; however, it is not guaranteed and thus the model may misclassify

instructions, leading to further problems during protocol generation. This misclassification issue applies to all machine learning methodologies, and the high F-score score of the model used in this project suggests that misclassification is rarely an issue.

Another caveat to using a machine learning model is that it has to be updated and retrained if new categories want to be added. The Naïve Bayes model was partly selected due to the ease with which it can be updated to accommodate more categories. Since it only relies on the frequency of words and the categories they belong to, updating the model with a new category is simply determining the frequencies of words in the new category and updating the model with new training data just alters the given frequencies for the category it is in. In the case of this project, all of the containers provided by the Opentrons API are not accounted for as some are not commonly used in Biomedical Engineering laboratories. If a user wants to extend the model to be able to include a new container, this would require retraining of the model and the user would have to know how to do this. Thus, a potential future direction of this project would be to extend the program to allow the user to add in new own categories and training data, catering the model to lab specific uses. This feature would make the model customizable and adaptable to a user's particular speech patterns and containers, as well as generalizing the model for performance on different robotics platforms.

Conclusions and Future Directions

Currently, the program does not allow for control of the hot, cold, or magnetic plates. As a work around, the user can just turn on the heated or cooled plate and place well plates on top, allowing for heating and cooling of samples. It would greatly enhance the applications of the project if control over a switch to turn on the hot and cold plates could be implemented. Furthermore, this addition would be an extension to the current Opentrons API as hot and cold plate control is not offered and requires manual control by the user. The magnetic plate control is offered through the current API, however a way to engage and disengage the hardware was not implemented due its highly specialized use case. Adding in the hardware modules to the voice control would give the user more capabilities to create complex protocols and would make an excellent extension of the project.

Further extensions include error detection and automatic checking for container placement. The predesigned 15 milliliter and 50 milliliter conical flask holder only allows one type of conical tube in each well. A future direction for this project would be to include logic to automatically know where 15 and 50 milliliter conical go and fill them up accordingly, similar to how the robot removes tips from a tip rack. This feature would reduce the complexity for the user when giving initialization instructions and would thereby reduce the total amount of time required to create a working protocol. Error detection would include alerts for the user in the event that an instruction they provided conflicts with a previous one, or if the wording of the sentence prevents information extraction. There is currently nothing in place to catch errors like this, so the user is responsible for reading over the protocol to make sure it seems okay. This is obviously not ideal and would be easier for the user to correct the mistake in the moment, however current protocol generation code compiles at the end after all instructions and not instantaneously as the steps progress. This method

is completely feasible and would require a slight, but not significant modification to the current code.

This project allows for complete protocol generation through verbal commands only, allowing an inexperienced programmer or any researcher who does not wish to code, to create useful Python protocols for the Opentrons robotics platform. The methods developed to achieve this goal are generalizable and are not Python specific, nor are they Opentrons specific. The action of pre-categorizing phrases greatly reduced the complexity of the text processing and made information extraction more accurate. Furthermore, this project increases the accessibility of science to those who may lack the fine motor coordination for traditional wet lab bench work, while still enabling independence without a great reduction in capabilities. Voice control platforms are becoming more widespread in society and thus this project fits right into that trend, extending an initially commercial concept into the realm of science, allowing for faster data generation and higher quality science by making incorporation of automation in the laboratory easy for any sized lab.

References

1. Chapman, T. Lab automation and robotics: Automation on the move. *Nature* **421**, 661–666 (2003).
2. King, R. D. *et al.* The Automation of Science. *Science* **324**, 85–89 (2009).
3. Begley, C. G. & Ellis, L. M. Drug development: Raise standards for preclinical cancer research. *Nature* **483**, 531–533 (2012).
4. Jessop-Fabre, M. M. & Sonnenschein, N. Improving Reproducibility in Synthetic Biology. *Front. Bioeng. Biotechnol.* **7**, (2019).
5. Haile, S. *et al.* Automated high throughput nucleic acid purification from formalin-fixed paraffin-embedded tissue samples for next generation sequence analysis. *PLOS ONE* **12**, e0178706 (2017).
6. Opentrons. Available at: https://opentrons.com/about. (Accessed: 15th September 2018)
7. Protocol Designer | Opentrons App Help Center. Available at: https://support.opentrons.com/protocol-designer. (Accessed: 17th September 2018)
8. Hutchins, J. From First Conception to First Demonstration: the Nascent Years of Machine Translation, 1947–1954. A Chronology. *Machine Translation* **12**, 195–252 (1997).
9. Hirschberg, J. & Manning, C. D. Advances in natural language processing. *Science* **349**, 261–266 (2015).
10. de Marneffe, M.-C., MacCartney, B. & Manning, C. D. Generating Typed Dependency Parses from Phrase Structure Parses. 6 (2006).
11. Manning, C. *et al.* The Stanford CoreNLP Natural Language Processing Toolkit. in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* 55–60 (Association for Computational Linguistics, 2014). doi:10.3115/v1/P14-5010
12. Berant, J., Chou, A., Frostig, R. & Liang, P. Semantic Parsing on Freebase from Question-Answer Pairs. in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* 1533–1544 (Association for Computational Linguistics, 2013).
13. Bird, S., Loper, E. & Klein, E. *Natural Language Processing with Python*. (O'Reilly Media Inc., 2009).
14. Falk, V. *et al.* Robot-assisted minimally invasive solo mitral valve operation. *The Journal of Thoracic and Cardiovascular Surgery* **115**, 470–471 (1998).
15. House, B., Malkin, J. & Bilmes, J. The VoiceBot: a voice controlled robot arm. in *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09* 183 (ACM Press, 2009). doi:10.1145/1518701.1518731
16. Ruurda, J. P., van Vroonhoven, T. J. M. V. & Broeders, I. A. M. J. Robot-assisted surgical systems: a new era in laparoscopic surgery. *Ann R Coll Surg Engl* **84**, 223–226 (2002).
17. Duvallet, F. *et al.* Inferring Maps and Behaviors from Natural Language Instructions. in *Experimental Robotics: The 14th International Symposium on Experimental Robotics* (eds. Hsieh, M. A., Khatib, O. & Kumar, V.) 373–388 (Springer International Publishing, 2016). doi:10.1007/978-3-319-23778-7_25
18. Hemachandra, S. *et al.* Learning models for following natural language directions in unknown environments. in *2015 IEEE International Conference on Robotics and Automation (ICRA)* 5608–5615 (2015). doi:10.1109/ICRA.2015.7139984
19. Pires, J. N. Robot-by-voice: experiments on commanding an industrial robot using the human voice. *The Industrial Robot; Bedford* **32**, 505–511 (2005).

20. Rogowski, A. Industrially oriented voice control system. *Robotics and Computer-Integrated Manufacturing* **28**, 303–315 (2012).
21. Matuszek, C., Herbst, E., Zettlemoyer, L. & Fox, D. Learning to Parse Natural Language Commands to a Robot Control System. in *Experimental Robotics: The 13th International Symposium on Experimental Robotics* (eds. Desai, J. P., Dudek, G., Khatib, O. & Kumar, V.) 403–415 (Springer International Publishing, 2013). doi:10.1007/978-3-319-00065-7_28
22. Mustafa, M. K., Allen, T. & Appiah, K. A comparative review of dynamic neural networks and hidden Markov model methods for mobile on-device speech recognition. *Neural Comput & Applic* (2017). doi:10.1007/s00521-017-3028-2
23. Cloud Speech-to-Text - Speech Recognition | Cloud Speech-to-Text API. *Google Cloud* Available at: https://cloud.google.com/speech-to-text/. (Accessed: 8th February 2019)
24. Watson Speech to Text. (2016). Available at: https://www.ibm.com/watson/services/speech-to-text/. (Accessed: 12th September 2018)
25. Cloud Speech-to-Text - Speech Recognition | Cloud Speech-to-Text API. *Google Cloud* Available at: https://cloud.google.com/speech-to-text/. (Accessed: 12th September 2018)
26. Mirończuk, M. M. & Protasiewicz, J. A recent overview of the state-of-the-art elements of text classification. *Expert Systems with Applications* **106**, 36–54 (2018).
27. Medhat, W., Hassan, A. & Korashy, H. Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal* **5**, 1093–1113 (2014).
28. Rish, I. An empirical study of the naive Bayes classifier. **3**, 6 (2001).
29. Pranckevičius, T. & Marcinkevičius, V. Comparison of Naïve Bayes, Random Forest, Decision Tree, Support Vector Machines, and Logistic Regression Classifiers for Text Reviews Classification. *Baltic Journal of Modern Computing; Riga* **5**, 221–232 (2017).
30. Hastie, T., Tibshirani, R. & Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. (Springer, 2009).
31. Quinlan, J. R. Induction of decision trees. *Machine Learning* **1**, 81–106 (1986).
32. Xhemali, D., Hinde, C. J. & Stone, R. G. *Naïve Bayes vs. Decision Trees vs. Neural Networks in the Classification of Training Web Pages*.
33. Frank, E. & Bouckaert, R. R. Naive Bayes for Text Classification with Unbalanced Classes. in *Knowledge Discovery in Databases: PKDD 2006* (eds. Fürnkranz, J., Scheffer, T. & Spiliopoulou, M.) **4213**, 503–510 (Springer Berlin Heidelberg, 2006).
34. Walt, S. van der, Colbert, S. C. & Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* **13**, 22–30 (2011).
35. Loria, S. *Simple, Pythonic, text processing--Sentiment analysis, part-of-speech tagging, noun phrase extraction, translation, and more.: sloria/TextBlob*. (2019).
36. Sokolova, M. & Lapalme, G. A systematic analysis of performance measures for classification tasks. *Information Processing & Management* **45**, 427–437 (2009).
37. Silva, R. M., Almeida, T. A. & Yamakami, A. MDLText: An efficient and lightweight text classifier. *Knowledge-Based Systems* **118**, 152–164 (2017).
38. Zinchenko, K., Wu, C. & Song, K. A Study on Speech Recognition Control for a Surgical Robot. *IEEE Transactions on Industrial Informatics* **13**, 607–615 (2017).
39. Lv, X., Zhang, M. & Li, H. Robot control based on voice command. in *2008 IEEE International Conference on Automation and Logistics* 2490–2494 (2008). doi:10.1109/ICAL.2008.4636587
40. Wagholikar, K. B. *et al.* Clinical decision support with automated text processing for cervical cancer screening. *J Am Med Inform Assoc* **19**, 833–839 (2012).

41. Oudin, M. Feedback and General Impressions Interview. (2019).
42. Limited, R. C. *PyQt5: Python bindings for the Qt cross platform UI and application toolkit*.