Project
Intelligent Robotics III
ECE 510, Spring 2020
Tyler Hull

H.13 Human Interaction with an Avatar

Visualized character that demonstrates emotions by reading facial gestures. Blueprints for Natural Language and Machine Learning to interact with an Avatar.

Introduction

Working in robotics often requires a huge investment in equipment. In addition, this equipment is often specialized and fragile. This creates problems for robotics education, as platforms for experimentation can be easily broken from a test program that moves motors out of bounds. This project was envisioned to be a solution to that problem for students working on robotic emotions or robotic heads.

Using an "Avatar" of the robot, we can write programs to show emotions, process natural language, and eventually output those programs to the real robot in the lab. This project is a first step toward these goals and provides a starting point for future students to expand upon.

Many options were considered for software and platforms in this project. Some should be explored further, and others should be pivoted away from. I'll do my best to distill the good and bad for the options I considered. Most software is pasted directly in the appendix of this document, but all the pertinent files are hosted in a github repository for future reference.

Software Used in Project

Getting setup with the correct software is always a challenge for taking over any project. I tried to minimize the number of files and packages that were used in this project so it is easy to pick up in the future and has minimal challenge for getting started. Below are the major software packages that were used and I discuss some tips on using the files created or getting setup with each.

- Adobe Illustrator (Creating Vector art for Avatar)
    - Adobe illustrator is a great program for graphic design, and I used it to work on the art for my avatar "Ella". It costs $25 a month for a student subscription for all the Adobe software packages (I think this is a really good value), but there are some free alternatives as well.
        - The PSU Computer Action Team (CAT) maintains a list of computers with access to Adobe Illustrator on its website. As far as I am aware, there is one in the 3rd Floor Computer lab that students can use anytime.
        - Inkscape is an open source alternative to illustrator and it can be downloaded and used for free. However, I think it's a little hard to use. You could definitely make it work with some time and a few tutorials though.
- Adobe Photoshop (Editing Avatar Art and Generating Images)
    - I also used Adobe Photoshop to work on the art for Ella. If you get the student subscription for Adobe you'll get this bundled in with the others, but there are some free alternatives as well.
        - The PSU Computer Action Team (CAT) maintains a list of computers with access to Adobe Illustrator on its website. As far as I am aware, there is one in the 3rd Floor Computer lab that students can use anytime (has photoshop and Illustrator).
        - GimpShop is an open source alternative to illustrator and it can be downloaded and used for free. However, I think it's a little hard to use. You could definitely make it work with some time and a few tutorials though.
- Adobe Character Animator (Some animations and rigging)
    - This software would also come bundled with the student subscription. It's great for rigging up an avatar (they call them puppets) and doing some easy animations with it. It will even read your

facial expressions and animate them on the puppet, show arm movements, walking, and other actions. However, it's not very clear how one can use this as a part of another application.
    - There may be scripting support at some point in the future, but for now… I think it's really just limited to getting a feel for animations and possibly exporting single frames of art to be used in OpenCV.
- Python 3.8 (Python 3.8 for GUI and OpenCV control)
    - Tried to use the newest version of python so there would be good support moving forward. Your mileage may vary if you are trying to use an older or newer version.
- OpenCV 4.2 (Used for Machine Learning and video app)
    - This was the newest version of OpenCV at the time of writing. Should continue to be available for some time in the future.
- wxPython (Cross Platform GUI for Python)
    - A GUI toolkit for python that is cross compatible with Window, Mac, Linux. So, it should work on your system once you get it installed. It is not fun to install on Linux. It is even less fun to install on a Mac running linux. I did detail some things in the readme and requirements.txt files posted to github that should help if you're unfortunate enough to be dual-booting on mac.
    - Might be best to update this to work with PyQt, but it works on wxpython for now.
- Amazon LEX (used for building conversational interface)
    - Amazon LEX is a Natural Language Understanding (NLU) and Natural Language Processing (NLP) managed service on the AWS platform. It's made to work with many of Amazon's other services and it's built on the Amazon Alexa backend. So this makes it useful for building conversational "bots".
    - It is a little challenging to get started with, but it has a lot of potential for interpreting and responding with audio files. This would be really cool for our Avatar so I suggest you explore it, but there are other tools you could use too. For instance, you could do all the NLP in your program and send it to Amazon Polly for text to speech.

Research into Controlling Animations with Python

There are many tools for creating animations or 3D images, but not a lot of tools for controlling those animations from a different program. This was one big challenge that I faced while working on this project. Below I highlight a few different options that I looked into. I'm sure that future groups will have other options available, so hopefully this description will provide a little insight into what might be good to look for when choosing a platform for their work.

Looking into Improv

Figure 1: Human2Machine Improv show with robots from Curious Comendy 2017

In 2017, I attended a Robot Improv show at the Curious Comedy Theater in Portland OR. This was before I decided to go back to school to pursue an engineering degree and I met Dr. Perkowski here (He encouraged me to come take his classes). At this show two machine learning and AI researchers, Piotr Mirowski and Kory Mattheson, were performing alongside robots they had developed. I recommend checking out their websites where they have many publications they have written and some really interesting projects.

Kory's project was a conversational avatar named Pyggy. When I took on this project, I thought back to that event and wondered if there were any tools we could use from that project to incorporate with our avatar. I was able to find details about it on his blog and his old files on his Github page.

Kory Mattheson, Research Scientist, Google (DeepMind)
https://github.com/korymath/Pyggy

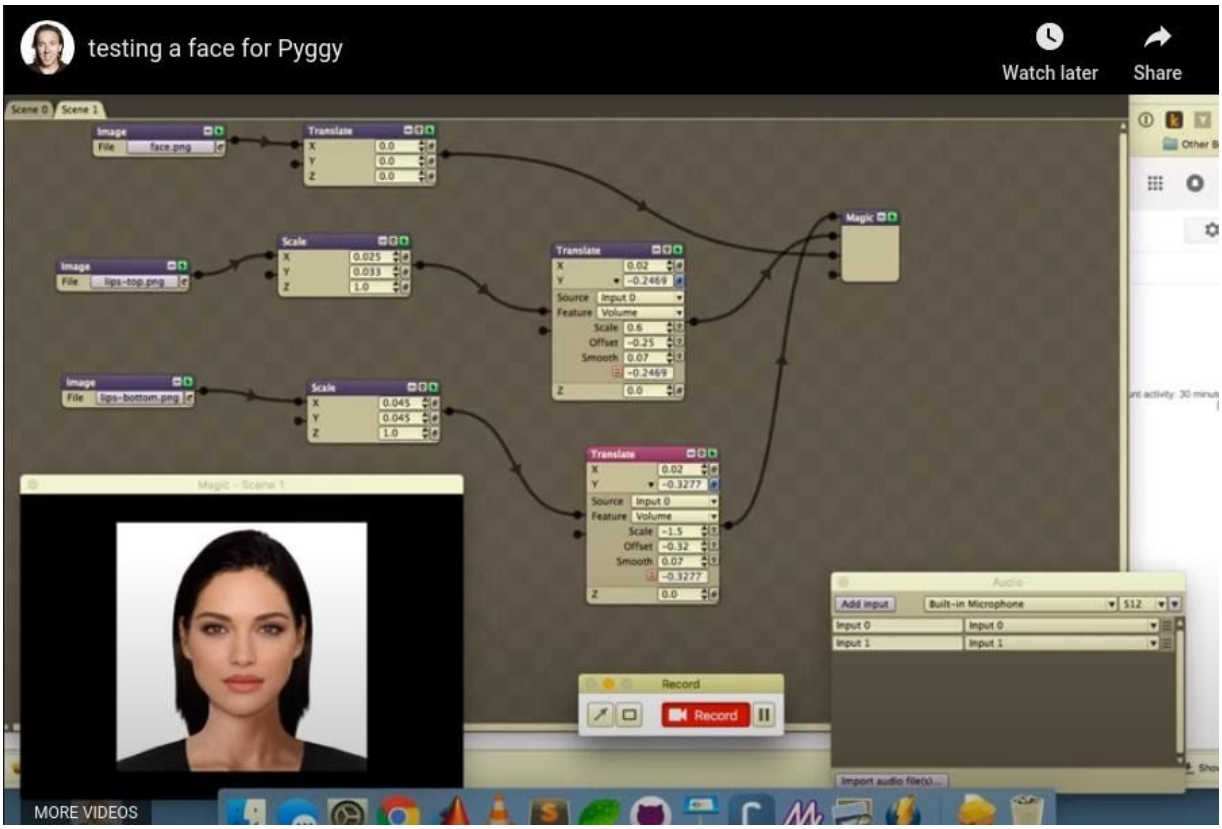
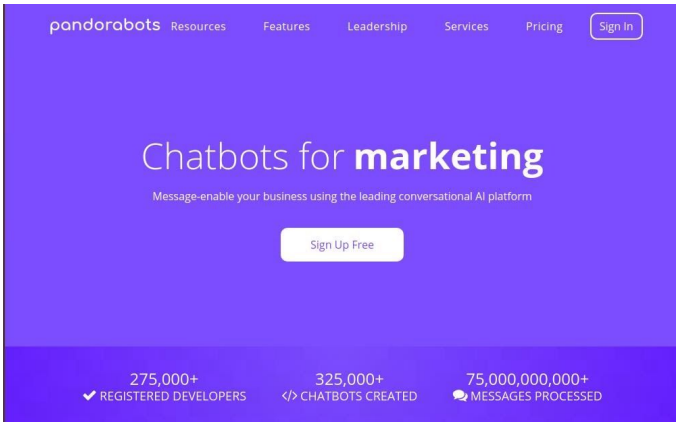
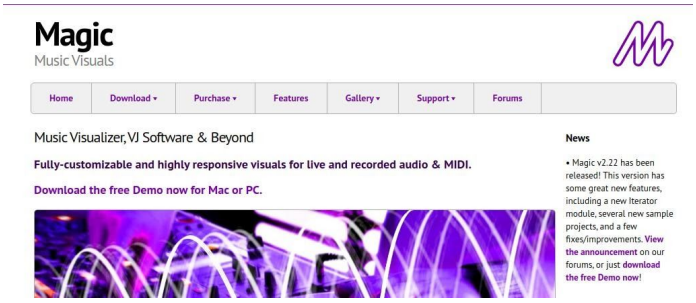


Figure 2: MAGIC DJ software interface for PYGGY Avatar.





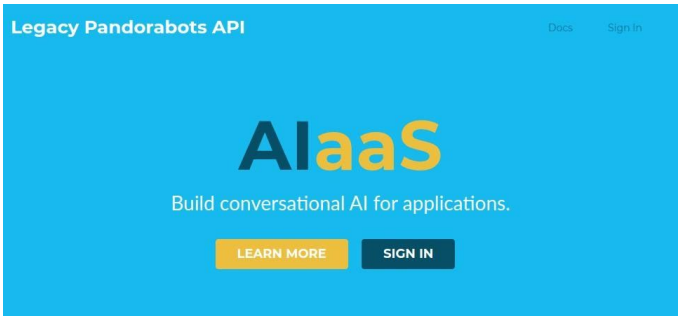


Figure 3: Chatbot services and APIs used for PYGGY Avatar project.

PYGGY Pros and Cons

Pros

- Open source
- Conversational chatbot working
- Already contains avatar images

Cons

- Avatar is very limited and uses DJ software for animation
- Uses PandoraBots and Artificial Intelligence Markup Language (AIML)
- Project is older and used some early API versions

Conclusion on Pyggy

The nice thing about a project like this is that it's open source, and looks to be mostly working. A really nice starting point for trying something out on your own. Unfortunately, this avatar is really limited and the mouth movements feature a raising and lowering box cutout for the chin and lower lip. It's also a few years old. So it appears that some of the chatbot company APIs it used have changed hands or it uses older APIs with less support. In addition, it uses a non-free DJ software for the visualization.

It's a great project, and you should look it over, but there isn't much we can re-use for a new avatar project.

Scripting In Blender

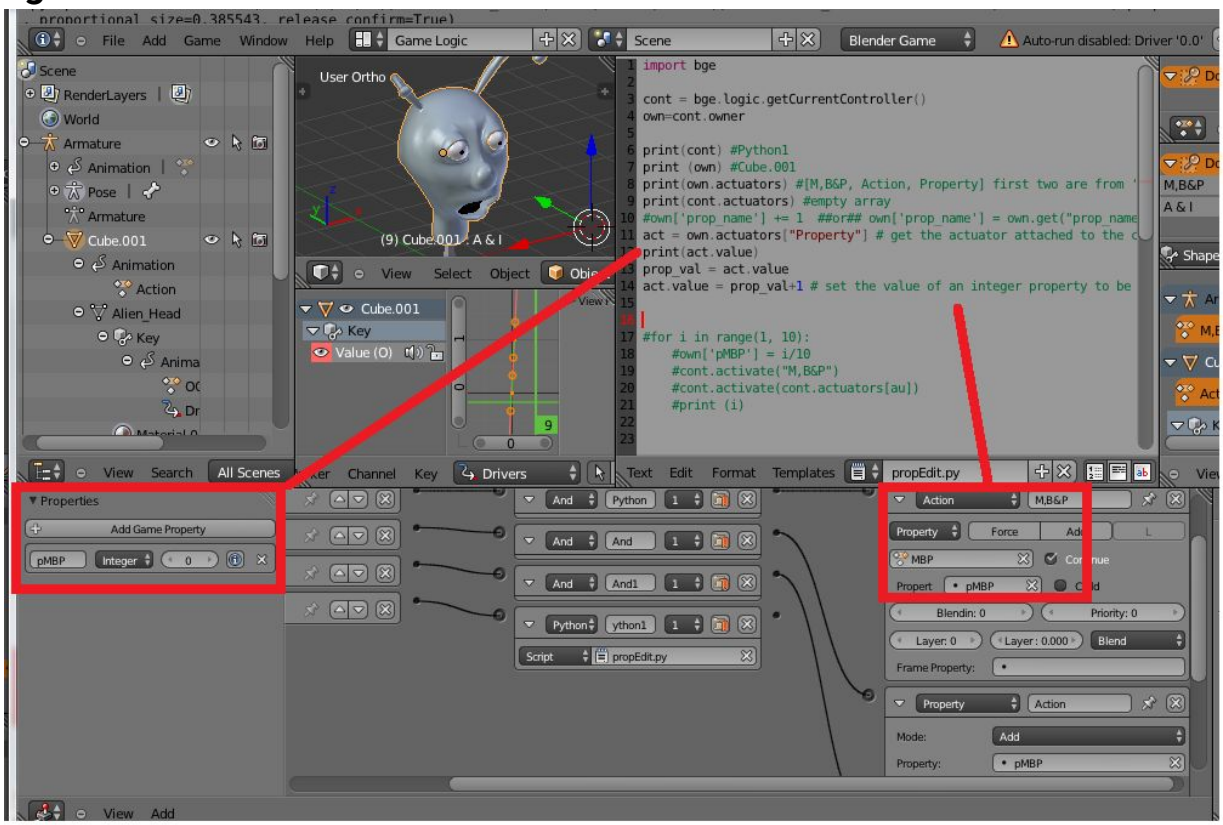


Figure 4: Blender Animation Interface with Scripting window for Python

Blender is an open source 3D design and animation software. It's very robust, complicated, expandible, and challenging to learn. However, it can make some really great animations and it has built in support for python. This probably makes it the best tool for what we are trying to do and I recommend that future teams dive in to this and recreate Ella as a model in Blender.

It is a little hard to find fully controlled animation projects, but there is a really cool project that modeled a monkey in Blender. What's even better is that they wrote a python library to control the model and they posted that to github.

Unfortunately, the model is not released as open source yet, but the library is available to be used. If I had found this project earlier in the term I would have tried to modify this or use it as a starting point for our avatar project.

Another possibility with Blender is that a 3D render of a robot could be taken to obtain the model and then rigged in Blender. Then we would have a model of a real robot we could animate as an avatar.

Adian Murphy, Primate Neuroscientist, Princeton
https://github.com/MonkeyGone2Heaven/MacaqueBlender/blob/master/README.md

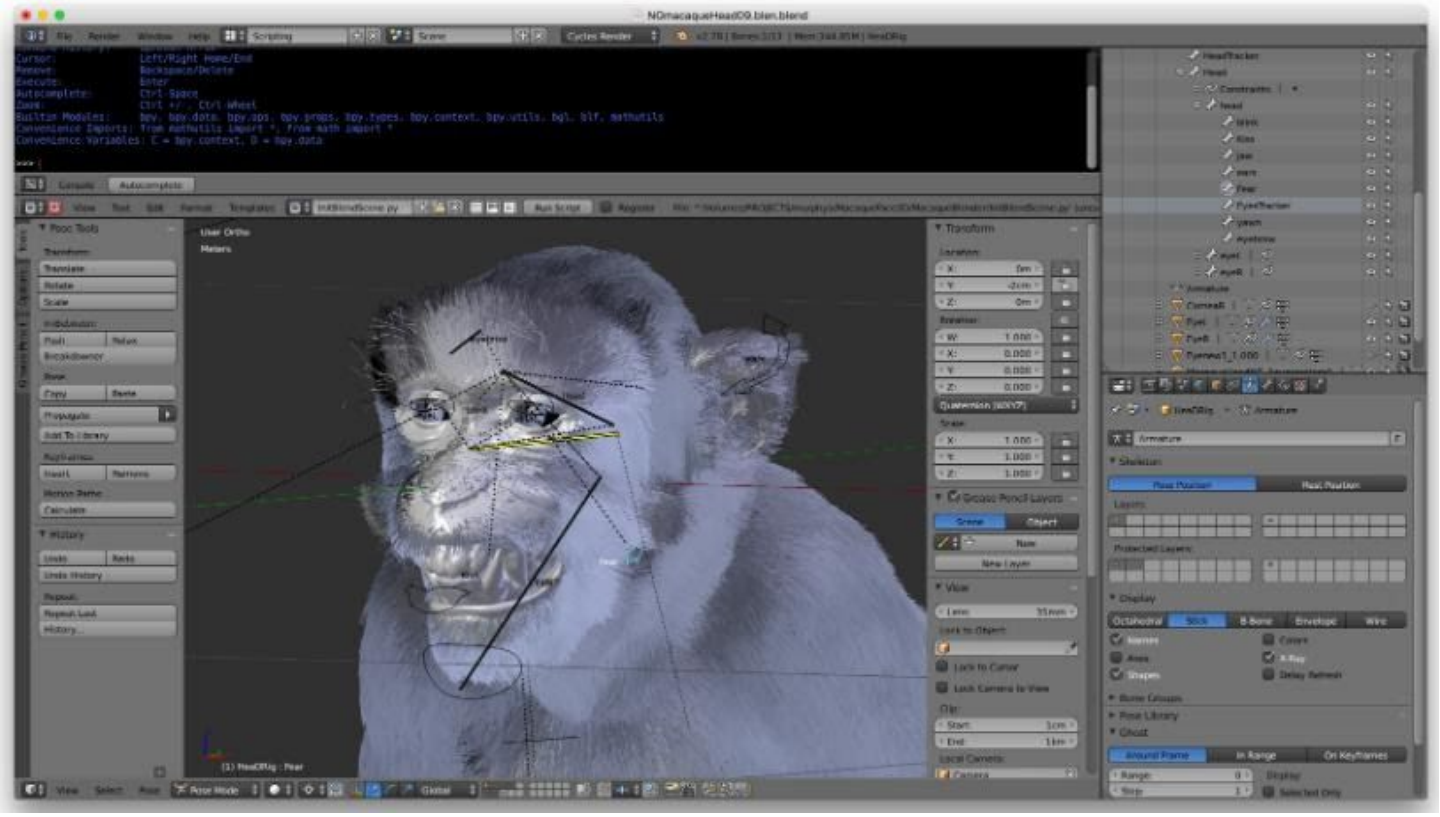


Figure 5: Professionally Rigged animation of a Macaque Monkey in Blender.

Animations created with MacaqueBlender project

Figure 6: Animations made with the Macaque Blender Model and controlled with a Python Library.

Blender Pros and Cons

Pros

- Open source animation and 3D software package
- Lots of community support and tutorials
- Python scripting built in for creation of animations or control

Cons

- Software is complicated and was unfamiliar to me
- Creating and rigging 3D animations is challenging
- Not a lot of information for controlling fully rigged models with Python

Conclusion on Blender

While Blender does seem to be a more complicated option for setting up and controlling an Avatar, it is the most versatile. I wish I would have chosen to dive in and explore Blender this term instead of using 2D art, but hopefully with a 3D scanner, some tutorials, and the Macaque Blender project above, future students will have an easier path to incorporating Blender animations into the Avatar project.

Avatar with Adobe Character Animator

Figure 7: Adobe Character Animator is one of Adobe's newest programs with facial movement tracking.

I am familiar with Adobe tools from other projects in the past, so Adobe character animator seemed like a great option for a starting point with animations. There are many avatars or "puppets" available for the software and it's easier to create joints, motions, triggers, etc in the software. You can setup certain features like hair to respond to gravity and mouth and facial expressions. Unfortunately, there isn't any support for scripting yet and while just about everything we want is built in… there is no clear way to use it in OpenCV.

Figure 8: Adobe Character Animator Stock Character showing face tracking control panel.

Lots of Pre-Rigged “puppets” and tutorials Available

https://okaysamurai.com/puppets/

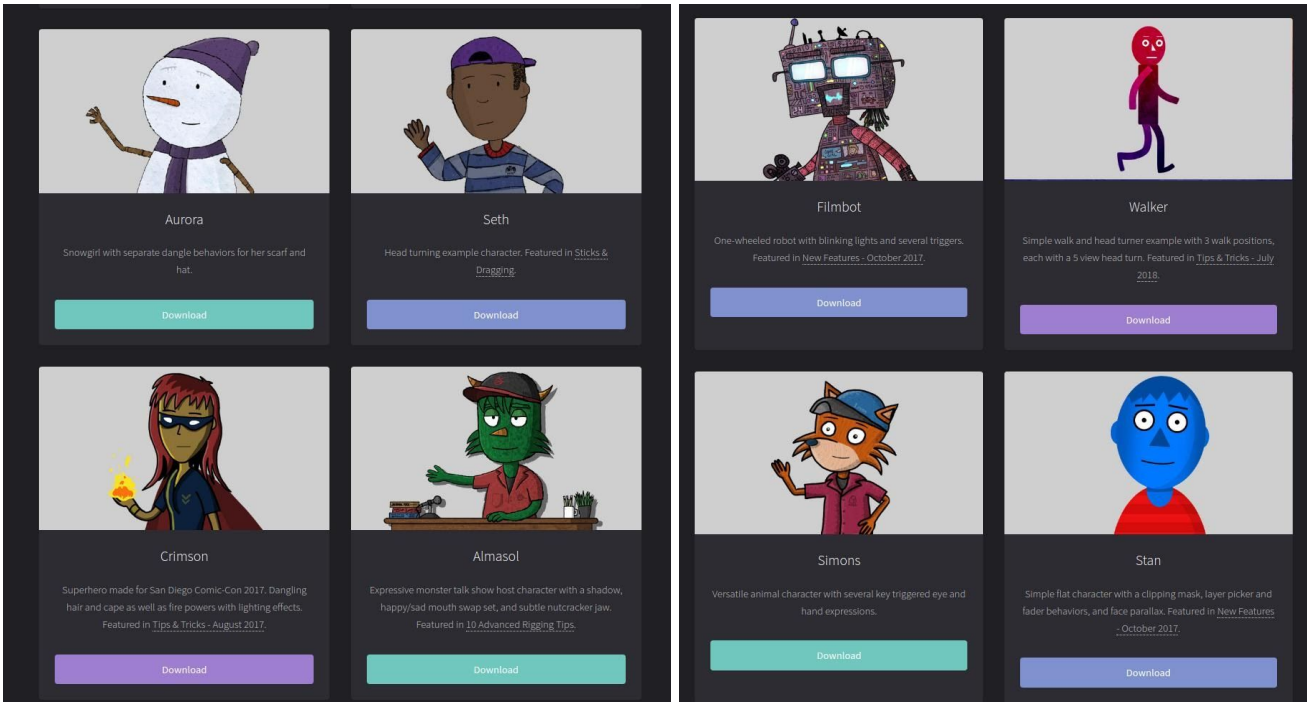


Figure 9: Collections of premade and pre-rigged characters called puppets are readily available.

Character Animator Pros and Cons

Pros

- Lots of tutorials available online
- Lots of premade Avatars or “Puppets” available for download
- Familiar tools that were similar to Adobe After Effects and Illustrator/Photoshop

Cons

- Closed Source and has monthly cost ($25/month for students - All adobe Apps)
- Needs to output a video or frame by frame images for use in OpenCV
- No Scripting built in, so it does internally what we want, but no software control

Conclusion on Adobe Character Animator

While it has many tutorials and art files available to use, as well as some cool built in features, it’s really only designed to generate videos or animations. This is great for exporting videos, but not so great for trying to respond on the fly to user input.

Perhaps in the future, with some scripting support, animations could be created on the fly with some parameters and then played with OpenCV. Alternatively, we could potentially find a method to provide input to OpenCV from OpenCV and output character animator’s animations in an OpenCV window. For now, I think this tool is limited to easy setup and structuring of scenes or art for taking static images or short videos for use in our program.

Deciding on Animation Software

Why I decided to go with Adobe Character Animator

- Thought this would simpler based on my experience with Adobe
- Misunderstood the complexities of trying to export and match up video clips
- At the time, I wasn’t aware that there was no scripting support available yet

At the time of making the decision, I decided that using Adobe Character Animator was the best choice since I already had a subscription, familiarity with Adobe tools, and wanted to start in a simple way to ensure that I could make some progress during the term.

Originally, I intended to create many small videos of my character and play them in OpenCV as different inputs were received from the user. However, this made the program less responsive and challenging to orchestrate. I eventually decided to use static art and do on the fly animations at 10 frames per second. This stop motion style animation gave me a lot more options, though it took longer to create the art for this.

If I had it to do over, I would have chosen to use Blender and dove into learning the software.

Figure 10: First Character I choose to work with in Adobe Character Animator.

I somewhat arbitrarily chose this character for the first avatar since I thought it looked endearing, was premade, and included some other animations like a flame on its hands and moving hair. However, part way through the term we had a class discussion on the importance of choosing a character for a robot.

Avoiding scary characters and the “Uncanny Valley” were brought up several times. Also discussed was the difficulty in animating characters with latex skins so that the motions looked natural. However, when animating an animal character or puppet, motions are more forgiving and fabrics are easier to work with for attaching strings and gears.

This conversation led me to change the Avatar for this project and to try to find something more friendly. In addition, it became obvious that there was a need for finding an Avatar that could represent a real robot in the lab. So I wanted to find a path for the Avatar to become something that had a counterpart in the real world.

Think Like Disney or Jim Henson

Figure 11: Disney and Muppets characters used for inspiration in the design on the Avatar.

In order to find something that could work for our Avatar project long term, we need a robot that is friendly, posable, can be modified for our needs, isn't copyrighted, and has features that could be animated. So doing something in the spirit of Disney or Jim Henson was in order, but care should be taken not to copy other characters. A summary of key points from the conversation that became a guide for what our Avatar should look like is below.

Build the Avatar, so it Can also be the Real Thing.

- No Devil Character (Or other scary characters)
- No copyrighted characters (In the spirit of disney, not copied from disney)
- Characters with Fur don't require as precise of movements
- Eye movements and Eye Brows, really important for emotions animation.
- Next year, students working on robot heads

Working Backwards from a puppet

https://www.puppetsandprops.com/wp-content/uploads/2018/05/The-Fred-Project.pdf

Figure 12: Design for moving eyes and eyebrows, as well as a torso for the Fred Ventriloquist Puppet.

I started looking for robot head designs to get an idea of what kinds of mechanics a future robot might involve and a sense of the size needed. I happened upon a really interesting youtube video showing the mechanics of a ventriloquist dummy. These dummies have simple mechanical features that let the user move the eyes left and right, raise and lower eyebrows, and open and close the eyelids. The head can rotate and the chest leaves space for an arm or other mechanics.

These videos led me to a tutorial for building your own dummy, and gave me the idea to start looking at what puppets or dummies were available. Specifically, I wanted to find something that could easily be repurposed and filled with mechanics like the dummy above. Also, I wanted to find something with eyes that could be repurposed for animating and something that had eyebrows or could have them added later.

On amazon, I found a family of Yeti like puppets that met all the requirements for our Avatar. These puppets can be seen below in Figure 13.

https://www.amazon.com/Purple-Monster-Puppet-Ventriloquist-Style/dp/B004FRZRL6/ref=sr_1_37?dchild=1&keywords=Silly+Puppets&qid=1592098345&sr=8-37

Figure 13a: Puppets in the spirit of Disney or Jim Henson found on amazon.

I purchased the purple puppet for this project and this became the basis for the Avatar art I later named Ella. In addition to the puppet, I bought some purple zipper sets so that the puppet could be modified and resewn with zipper access later. This will allow the puppet to be used as a skin that can be easily removed and cleaned for a robot in the future. The puppet and zippers will be donated to the lab so that future students can bring Ella to life if they choose.

Figure 13b: Purple Yeti like puppet used as the basis for the Ella Avatar.

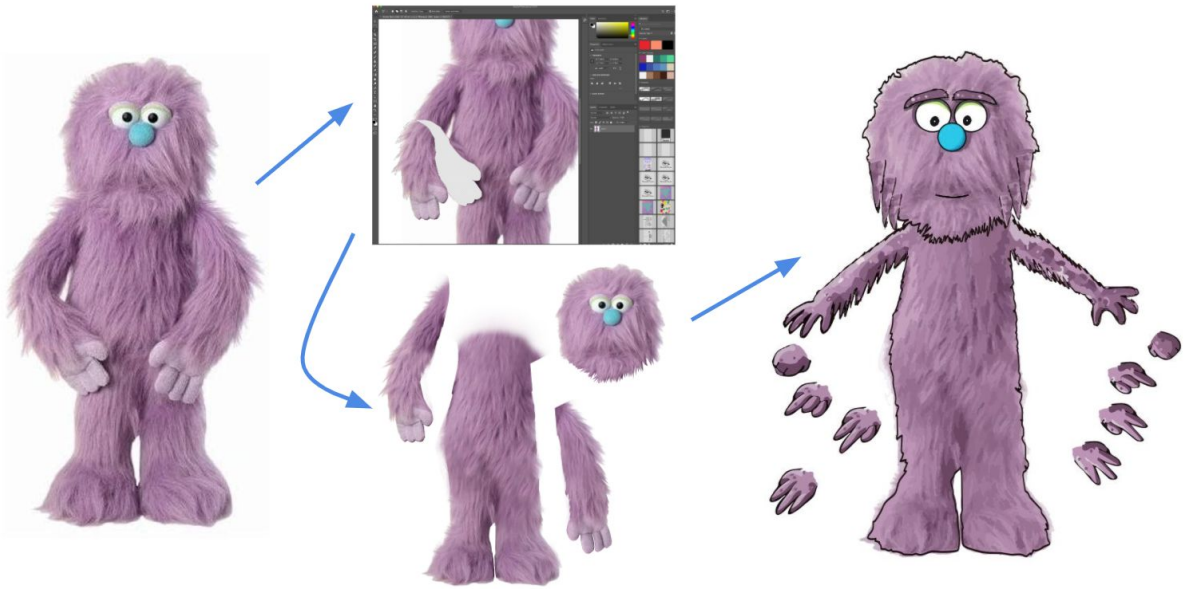


Figure 14: Creating a Adobe Character Animator puppet using Adobe Illustrator and Adobe Photoshop.

After deciding on the purple puppet, I used the image from the product listing to get started making the artwork for the avatar. This was done using photoshop and illustrator. My process for creating the art was to use the magnetic selection tool to select regions of the puppet and make them into layers.

For the arms, I selected a region where the hand was over the body and I did a free transform to rotate and position it away from the body. Then I used the stamp tool to fill in the area that was missing with sections from the rest of the body. Once this was done, I split the body into multiple parts and used the magnetic selection tool to separate the body parts from the white background.

Finally, I moved these images into illustrator and used the image trace function to convert the images to shapes. After this was done, I outlined the art in black. I ended up combining our new art with the arms from the original red devil puppet I chose above. Then I recolored the arms to purple to blend them in. This gave me the opportunity to use some functions for the other puppet in Adobe Character Animator like shoulder attachment points. It also gave me a starting point for some art for the hands. Then I created several more hand positions so the Ella can play Rock, Paper, Scissors or do math in the future.

Our Avatar “Ella” in Adobe Character Animator

Figure 15: Rigging and animating Ella in Adobe Character Animator.

Reading and Displaying Emotions

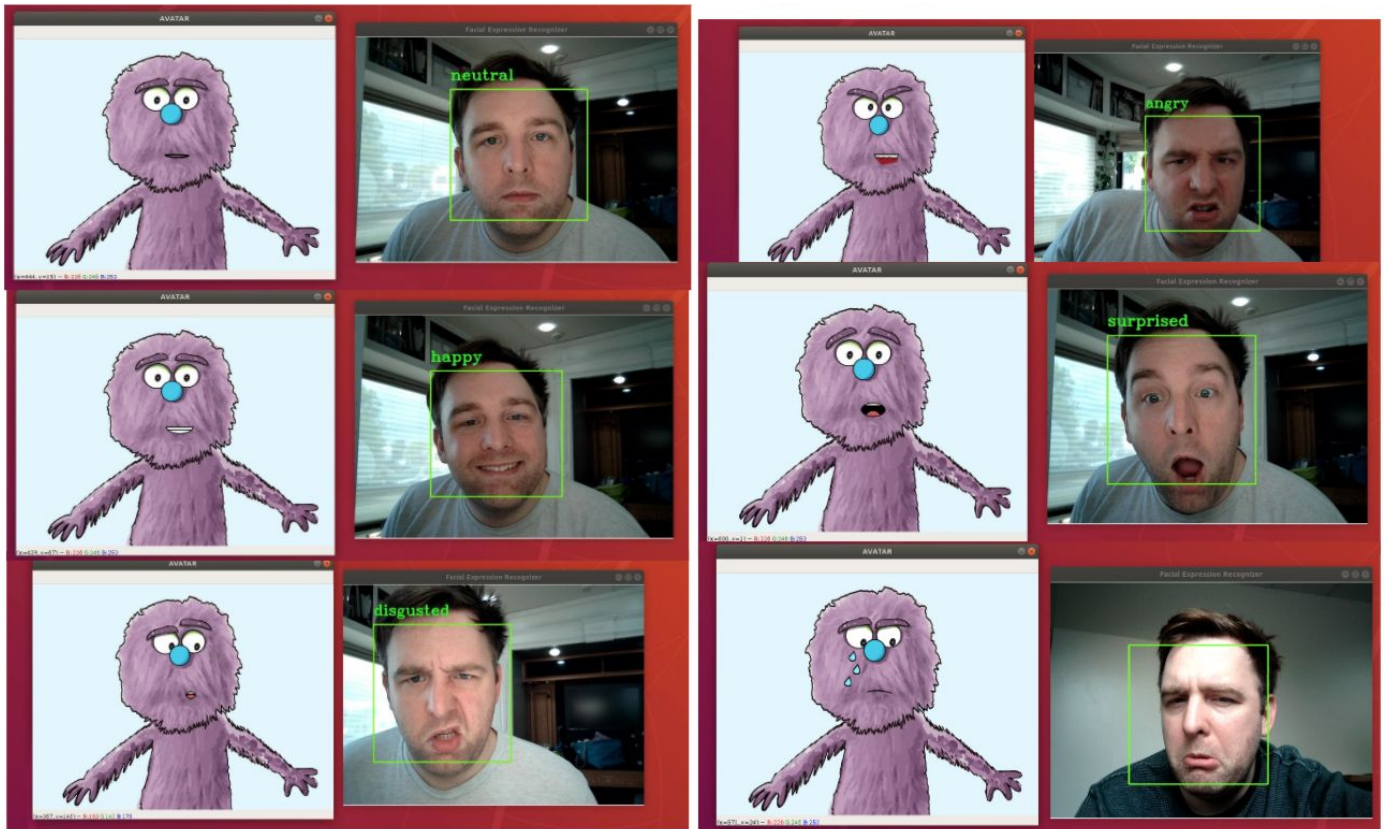


Figure 16: Facial emotions being expressed by the Avatar after live processing of a webcam frame in OpenCV.

Now that we have some art for our Avatar, it's time to start working on the interaction between the user and Ella. The first goal of this project was to create a system that would read the emotions of the user and have the avatar display the same emotion. In order to make this work we need our application to be able to do a couple of things. We need to be able to collect photos of the user and tag them with the correct emotion. We need to be able to train a model on that data. Finally, we need to be able to take camera frames, use the model to determine the emotion, and call an image for the avatar to display the same emotion.

Avatar Emotions App Overview

- Uses OpenCV built in Haar cascade classifiers (Viola-Jones)
    - There are many of these classifiers that are included with OpenCV. For this project, I use the face detection and eye detection classifiers. However, this requires me to take off my glasses. OpenCV also has a classifier for people who wear glasses, so this could be an option for future use in the program.
- Uses OpenCV built in Multi-level Perceptrons (MLPs)
    - An early, but still very useful method. These are built in to OpenCV and work very well. I was able to get a pretty reliable model trained on about 15-20 images for each emotion and the training completed in less than 10 min.
- Uses Principal Component Analysis (PCA) to reduce dimensionality of the data and speedup algorithm accuracy.
    - A large reason why we can train so quickly. This built in feature to OpenCV will reduce the complexity of our data so we don't have to process so much information. We can also select

how many features to look at since most of our data is in the first 20 or so features and we get diminishing returns for looking at more.

- Setup to read 6 emotions, but could be expanded. (Neutral, happy, sad, angry, surprised, disgusted)
    - We're set up to read only 6 emotions right now. However, you only need to add more to the radio buttons in the interface and to the process.py code if you want to add more. This lets you collect different emotions, tag them, process them, then read the tags back during the demo for selecting the correct avatar image to display.
    - I've already added extra radio buttons for capturing phonetic mouth shapes. This will hopefully allow Ella to show realistic mouth movements when audio is being played in the future. For now, she just mimics the mouth shapes of the user when they talk. It's a little buggy still, but should work better after some improvements to when the avatar reacts to a given tag or label.

Haar Cascade Classifiers in OpenCV

- Detects differences between light and dark areas in grayscale images
- Many different options built into OpenCV for different features
- For this project we're using haarcascade_frontalface_default.xml and haarcascade_eye.xml
- Need to convert our image to grayscale before we can run.

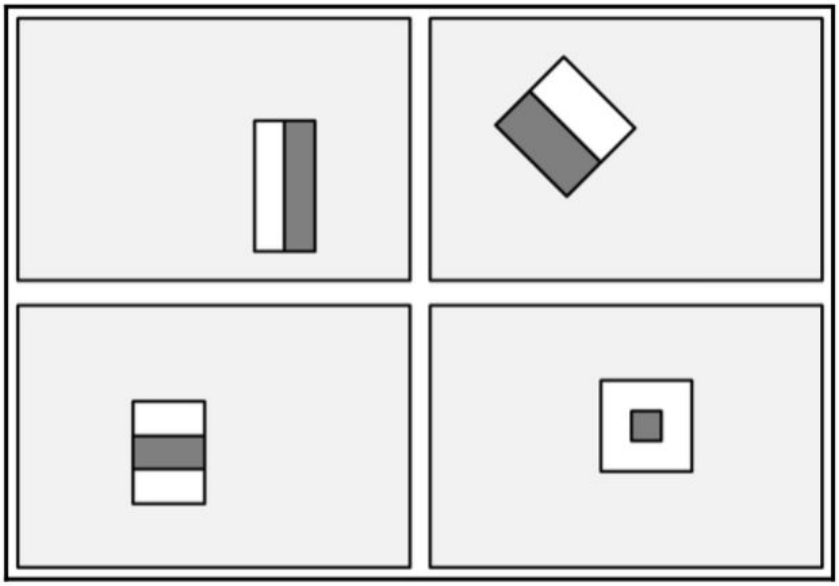


Figure 17: Illustration of light and dark shapes that are analyzed in a Haar Cascade Filter in OpenCV.

Haar classifiers work by taking a grayscale image and analyzing the orientation of light and dark shapes. In the images above we could imagine that the computer might see an eye as being similar to the shapes in the bottom right corner. A dark shape in the middle representing the pupil surrounded by a white shape for the cornea. Similarly, we can imagine that the whole eye area would be dark in a photo, with a bright spot on the cheek underneath. In this way, the Viola-Jones algorithm, which the Haar cascade classifier is based on, can determine various shapes and what they represent in the real world. Created not all that ago in 2001, this algorithm made it possible to efficiently do face recognition in real time.

Detect a Face & Color Gray

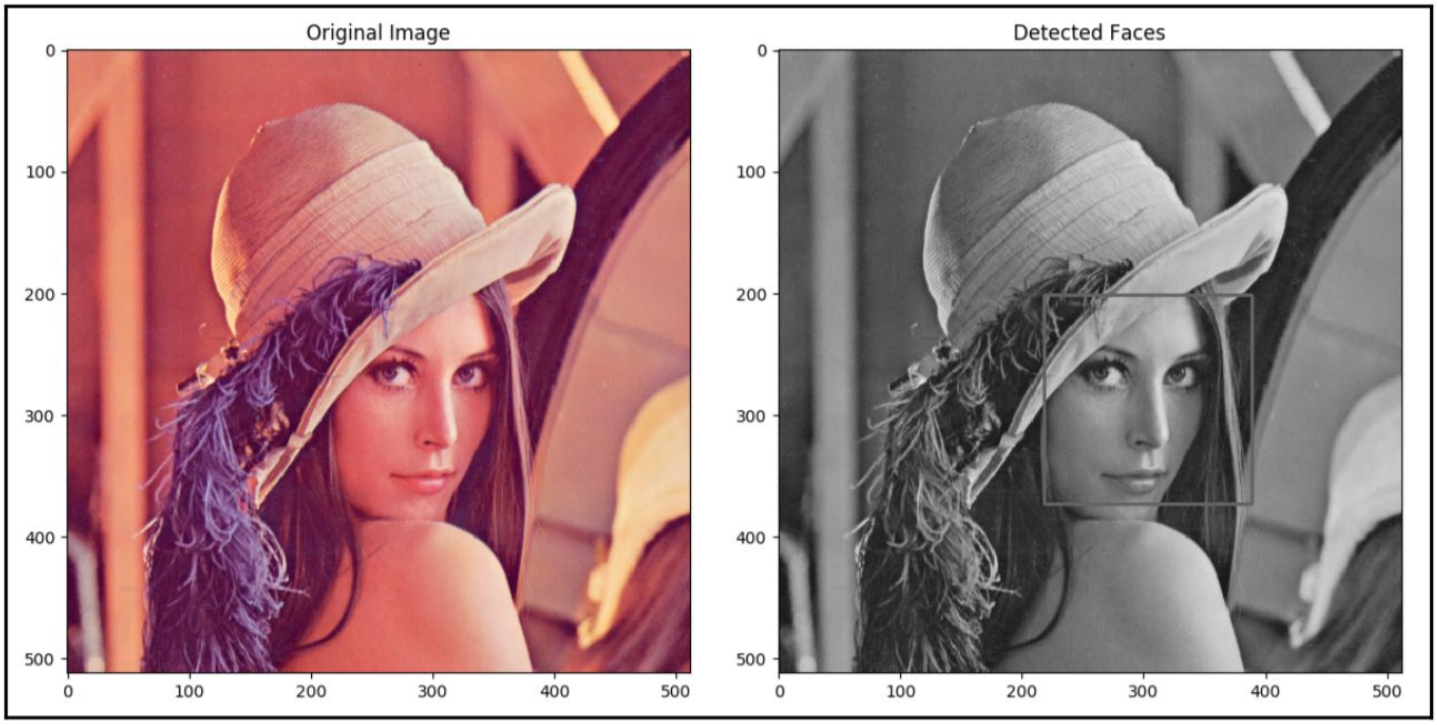


Figure 18: Example of face finding in OpenCV on the famous Lenna image.

In our images of Lenna above, you can see the process for how our avatar program processes images for use with this algorithm. First we take an image from the webcam stream. Then we convert it to grayscale like in the code snippet below. Next we can use the OpenCV commands to detect a face using the haarcascade_frontalface_default.xml file. This results in Lenna's face being detected in the frame in the image on the right.

However, this creates some additional headaches for us because we need to be able to see the face from a front view for this classification to work. Since we're taking images from a webcam stream where the user is moving, we need to do some transformations to help us give the classifier the correct data.

Detect a Face & Color Gray

```
import cv2

gray_img = cv2.cvtColor(cv2.imread('example.png'), cv2.COLOR_RGB2GRAY)

cascade_clf = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
faces = cascade_clf.detectMultiScale(gray_img,
                                     scaleFactor=1.1,
                                     minNeighbors=3,
                                     flags=cv2.CASCADE_SCALE_IMAGE)
```

Detect Eyes & Transform Face

```
# we want the eye to be at 25% of the width, and 20% of the height
# resulting image should be square (desired_img_width,
# desired_img_height)
desired_eye_x = 0.25
desired_eye_y = 0.2
desired_img_width = desired_img_height = 200

try:
    eye_centers = self.eye_centers(head)
except RuntimeError:
    return False, head

if eye_centers[0][0] < eye_centers[0][1]:
    left_eye, right_eye = eye_centers
else:
    right_eye, left_eye = eye_centers

# scale the distance between eyes to desired length
eye_dist = np.linalg.norm(left_eye - right_eye)
eyeSizeScale = (1.0 - desired_eye_x * 2) * desired_img_width / eye_dist
```

In the code above, you can see the work around used to let us use only the frontal face cascade multiplier. Here we use the eye detection classifier to find the position of the eyes. Then we create a square image where the eyes are a particular distance from the edges of the image. Our new square image can be used for our training and goes through a little transformation to try and keep the view straight on.

However, it's not a perfect solution. During data collection, we see some print statements in the terminal to let us know if the image met these qualifications and could be created. Otherwise, it displays a message that there was an error and we have to take another picture for the dataset.

This also extends to our real time detection. If the user has turned to the side while the demo is running, we can no longer determine what the emotion is. The tag is no longer displayed until the user moves again and we can detect a new emotion. This has some very real ramifications for the avatar art and required me to create a global last state variable for the emotions. This allowed the last detected emotion to continue to be shown on the avatar screen and kept jitter from happening due to the varying user input stream.

This solution works pretty well, especially since the avatar is fairly simplistic at the moment. In the future, it would be good to include additional classifiers so that a facial expression could be detected from a variety of angles. This would let the application be a little more responsive and possibly run at higher frame rates.

Running Data Collection

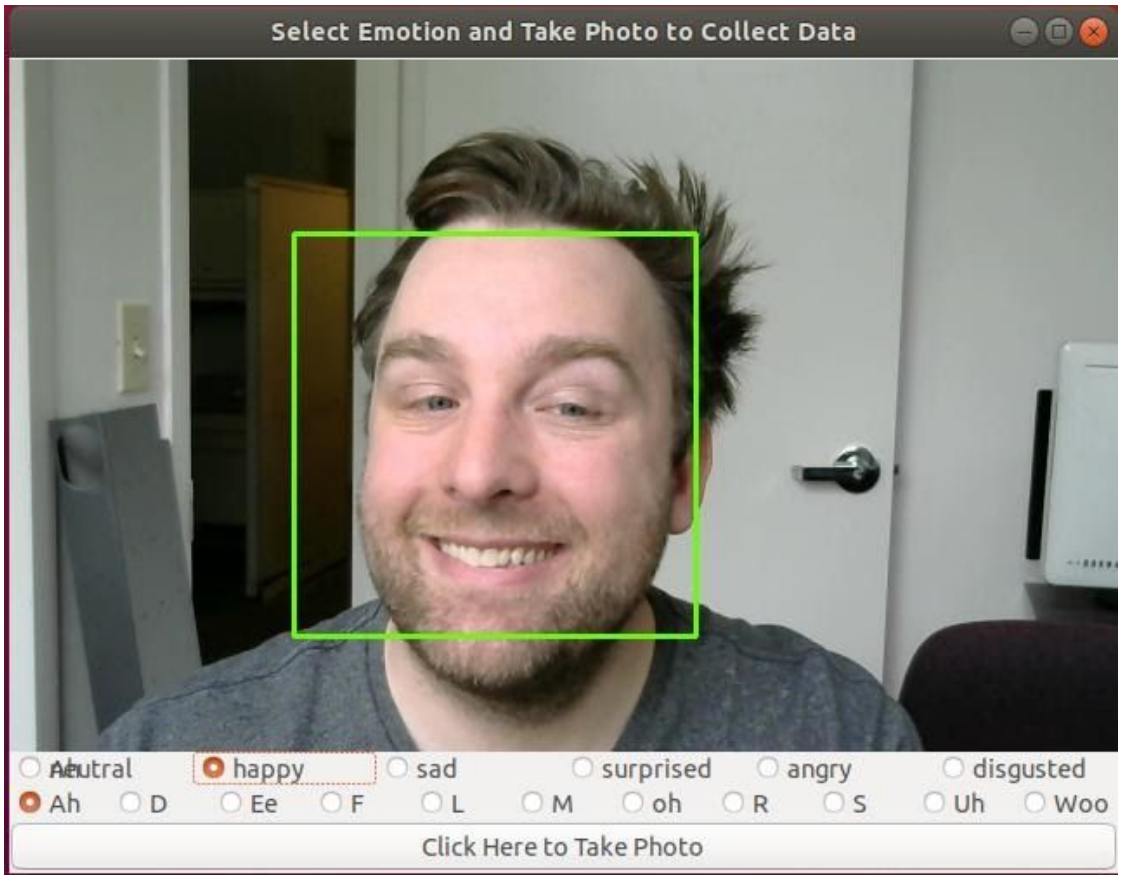


Figure 19: GUI display when running the data collection portion of the program, this older version shows a layering error, but includes options for phonetic mouth shapes that can be used to simulate talking.

The program uses an argument parsing method to determine which mode it should be running in. In Collection mode, we select the desired emotion to be collected and click the button to capture the image for processing. If the image met the criteria for processing, it's data is added to a .csv in the data folder that we can use to train our model on in the next step.

We can run the collection mode with something like the following command:

```
Python3 avatar_emo.py collect
```

Running Training

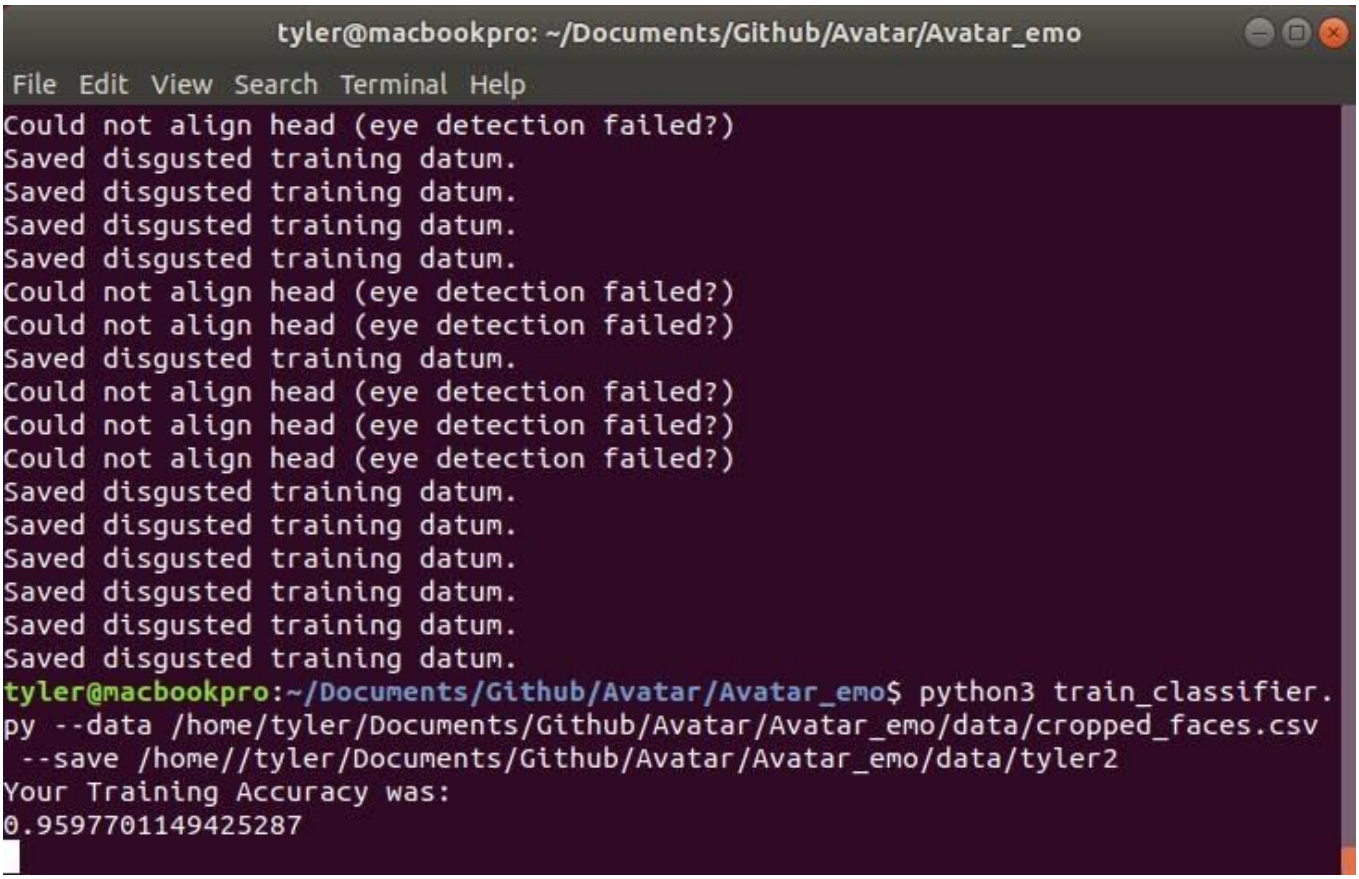


Figure 20: Output when running the training portion of the program.

Next we want to train our model on the images we just took of ourselves. We need to run the training program and provide it with the .csv where all the image data is stored. We can also provide it with a save location so that our new model can be stored and used in the future.

After the training completes, you are given a training accuracy to give you an idea of how good your model is. It's not a great representation of how it will perform, but it does give you a sense of if your images were decent or not.

One thing to look out for is the lack of print statements as this is training. It just shows as a black terminal while it works, then it prints out the accuracy after a few minutes. Unfortunately, after this, it's still processing the PCA features. Some additional print statements could be added to this to improve the user experience and let the user know what is going on. Otherwise, they may stop the training thinking that the program crashed or wasn't working. (I did this myself several times)

We can run the training mode with something like the following command:

`Python3 train_classifier.py --data <fullPathToData> --save <fullPathToSaveFolder>`

Running The Demo

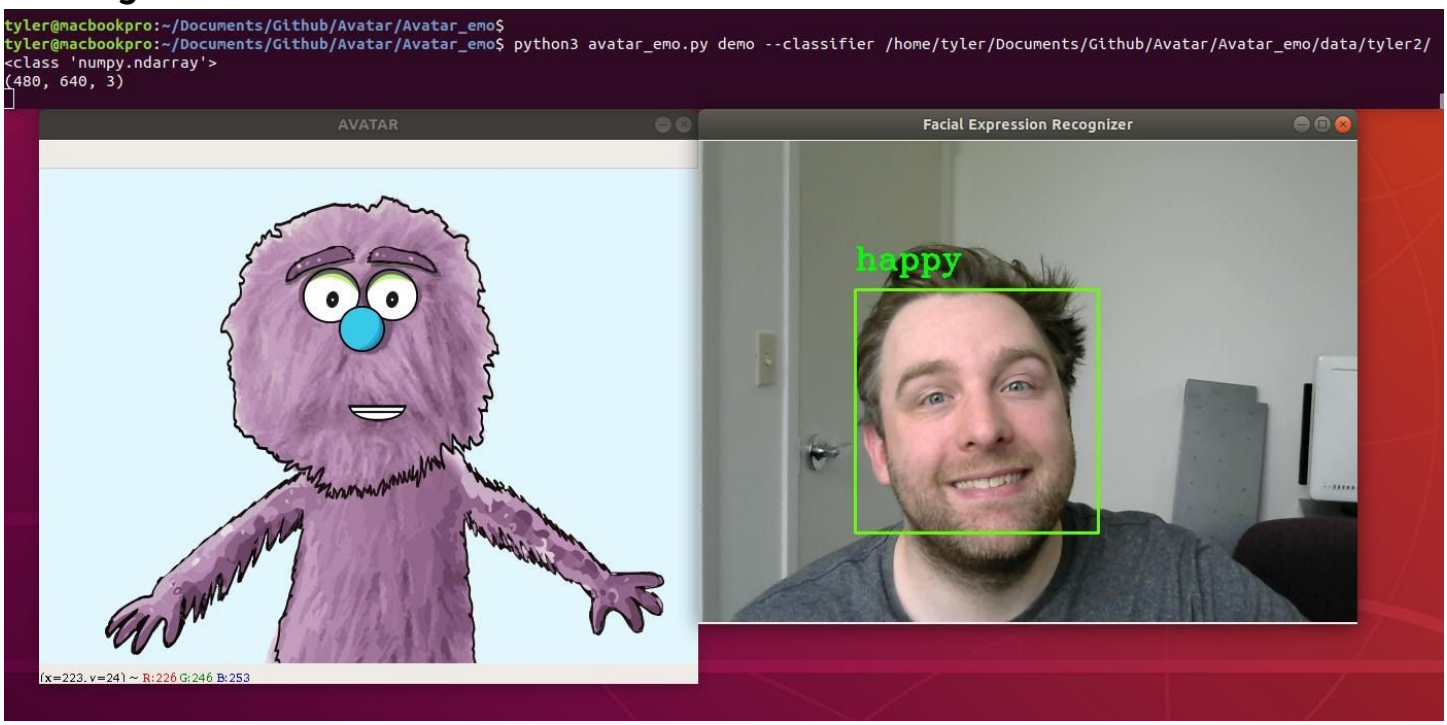


Figure 21: GUI display when running the DEMO portion of the program.

Now it's time to run the demo. We need to provide the classifier that was generated from our training code. If you give it the same path that you used for the place to save your training it will be able to unpack and use that model. Now we can show some emotion and see the response from the avatar.

We can run the Demo mode with something like the following command:

```
Python3 avatar_emo.py demo --classifier <fullPathToWhereYouSaved>
```

Works on Drawings as well

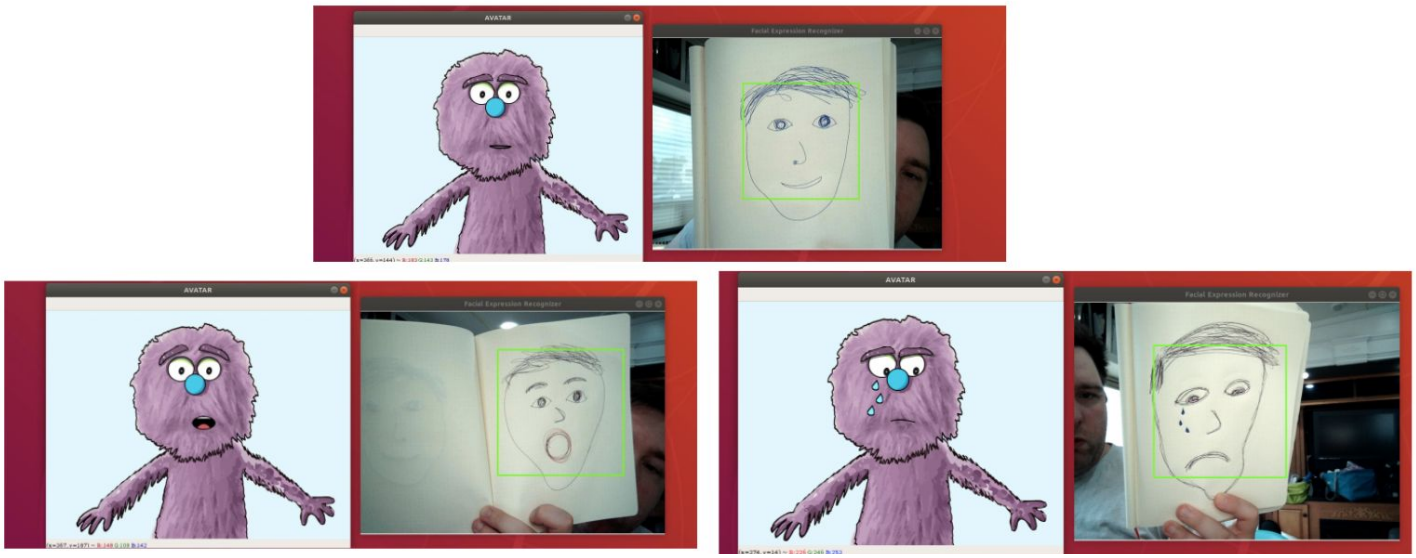


Figure 22: Emotion recognition capturing the face and emotion of a drawing.

As a test, I wanted to see if the application could read the emotions of images too or if it could only read my facial expressions. While it didn't have great results, it did reliably detect the face. The last state emotion kept the avatar from jumping and the program did capture the emotion from the drawing.

I didn't explore training with datasets that included multiple people, but this application should work on a large variety of users after some additional images are captured of different users.

Reading and Displaying Talking

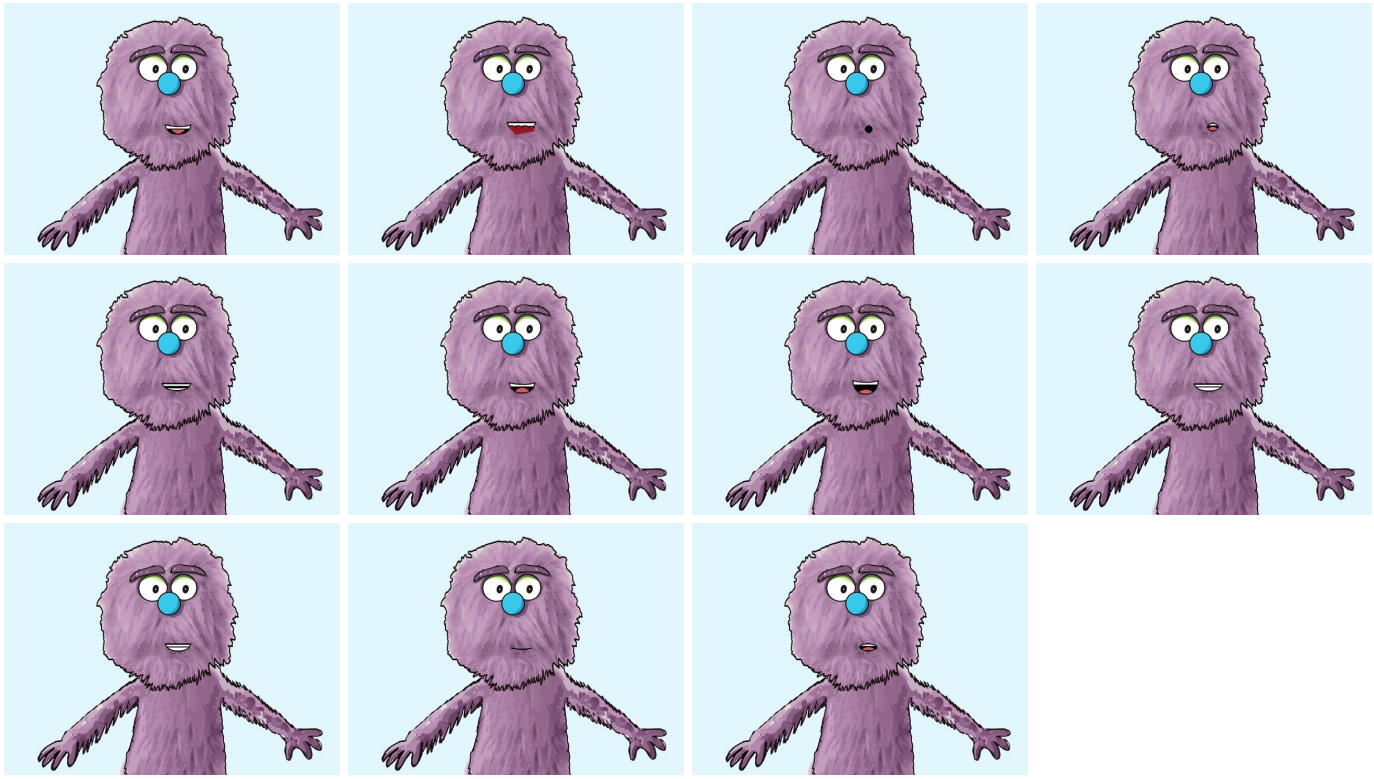


Figure 23: Illustrations of all of Ella's mouth shapes for common Phonetic sounds.

Figure 24: Phonetic sounds mouth shapes description from Adobe Character Animator.

- Mouth shapes for common phonetic sounds.
- Adobe Character Animator analyzes audio to translate sounds into these shapes.
- These positions were added as options for the collection mode in our emotions program.
- When trained and running on this model, our Avatar will look like it's saying the same things as the user.

In figures 23 above, you can see the avatar art for different phonetic mouth positions. We wanted Ella to be able to have an animated and realistic talking mode for when it's connected to an audio source. In this implementation, no audio is analyzed, but the model is trained on these mouth shapes. That way the avatar will be able to look like saying the same thing the user is saying.

Eventually, it would be good to connect this with some sort of text parsing or audio processing to make the avatar's speech look realistic for all types of inputs. This feature is a little buggy still in this application, but it could easily be improved upon in the future by additional training or mode selection in the program.

In the future, it should be possible to analyze a string of text and display these mouth movements as audio is played from Text To Speech is played back. A more complicated method might be to find an audio analyzation package for python and do this dynamically based on the audio files.

NLP and Machine Learning

Figure 25: Amazon LEX is an AWS service that provides developers access to the underlying Amazon Alexa technology.

What is Amazon LEX

- An AWS service for building conversational interfaces
- Uses Amazon Polly for Automatic Speech Recognition and Natural Language Understanding.
- Built on the same learning models as Amazon Alexa.
- Integrates with other services like Call Centers, Lambda Functions, Polly, Databases, and Rekognition.

In order to incorporate machine learning and Natural Language Processing, I attempted to use Amazon's AWS LEX service. This service uses the same backend that Amazon uses for it's ALEXA product that is running on smart home speakers and smart devices. Now developers have access to the same tools, but without the need to use the Amazon Alexa toolkit. This provides more flexibility in what kinds of projects can be developed with the service.

The purpose for this service is to make conversational chatbots like ALEXA. These conversational bots work by allowing us to define "intents". An intent is something that the chatbot can talk about. Once you have an "intent" setup, you can define "slots" for each intent. This gives the bot something it can remember and use later. So for example, an intent is to order pizza. A subject the bot can talk about. We give it a bunch of example phrases so it knows when we want to order a pizza. A slot in this case would be a topping we want on the pizza. This can be very transactional like a scheduler or ordering bot, but can be more robust if the bot can talk about family and remember who your parents are.

Example Program Flow

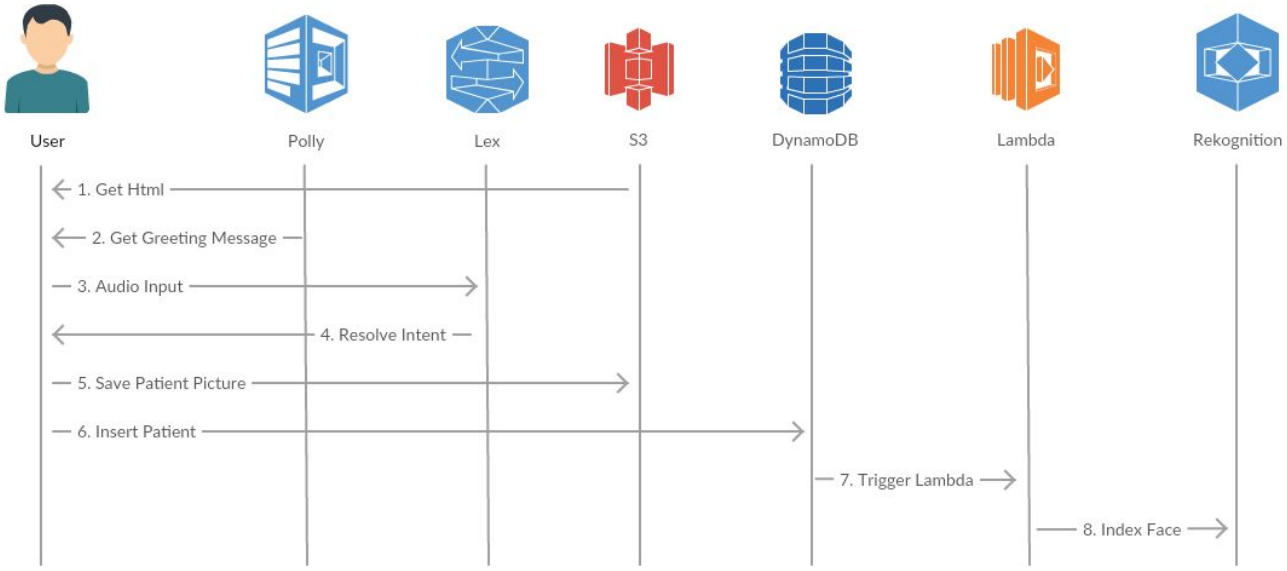


Figure 26: Program flow showing the interactions between various AWS services. More info on AWS machine learning services here.

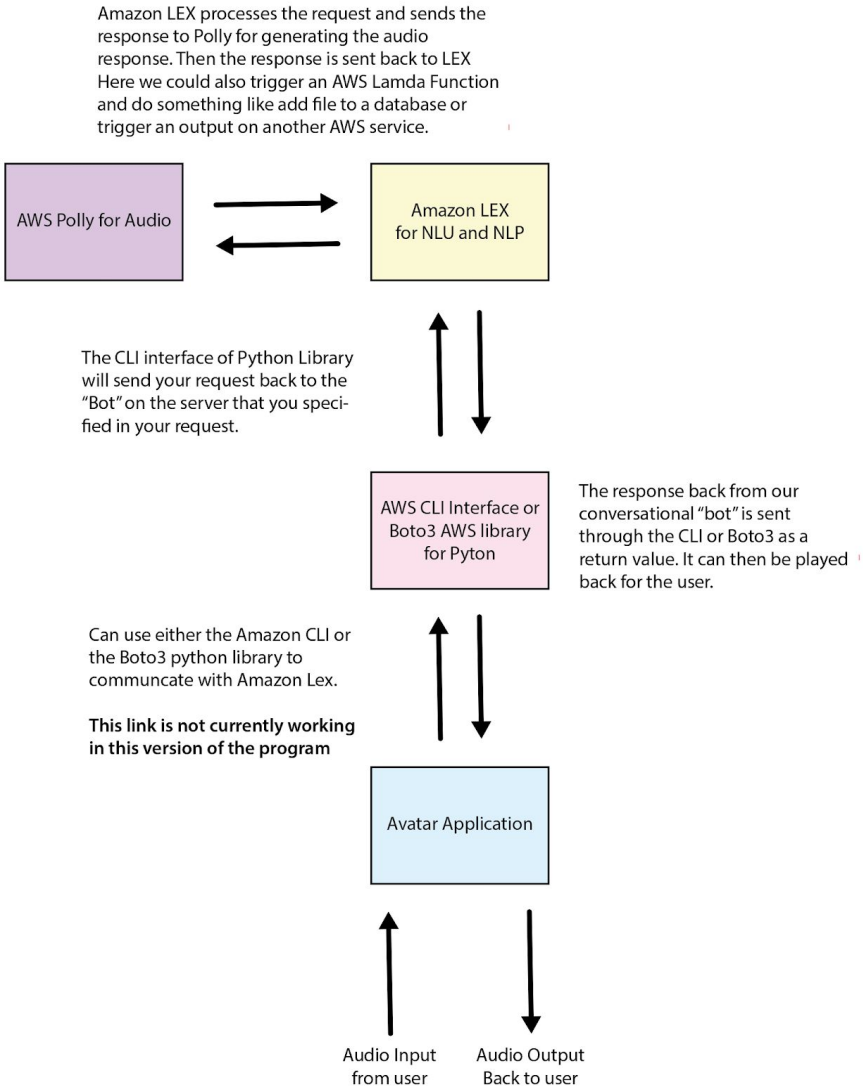


Figure 27: Program flow for this application. Unfortunately, the link between Amazon LEX and our python program still needs to be completed.

In figure 27 above, you can see a block diagram of the planned structure for this application. Unfortunately, I had difficulty with the connection between my AWS account and the python application. I'm not sure if the issue was with my AWS account or with the Boto3 library. There is also support for the AWS command line interface (CLI) and LEX, so it should be possible to send information on the command line from a python program to get this connection working.

Despite these issues, I was able to build a conversational bot in Amazon LEX and test it with the web interface. Below you'll see the setup for the bot that can have a conversation about scheduling classes for the summer term. It has many "intents" that were setup on the left side to allow for all kinds of phrases to be said that activate the bot. When the bot learns something about us, it stores that detail in a "slot". Below you can see that these slots can be used in phrases when the bot responds to the user.

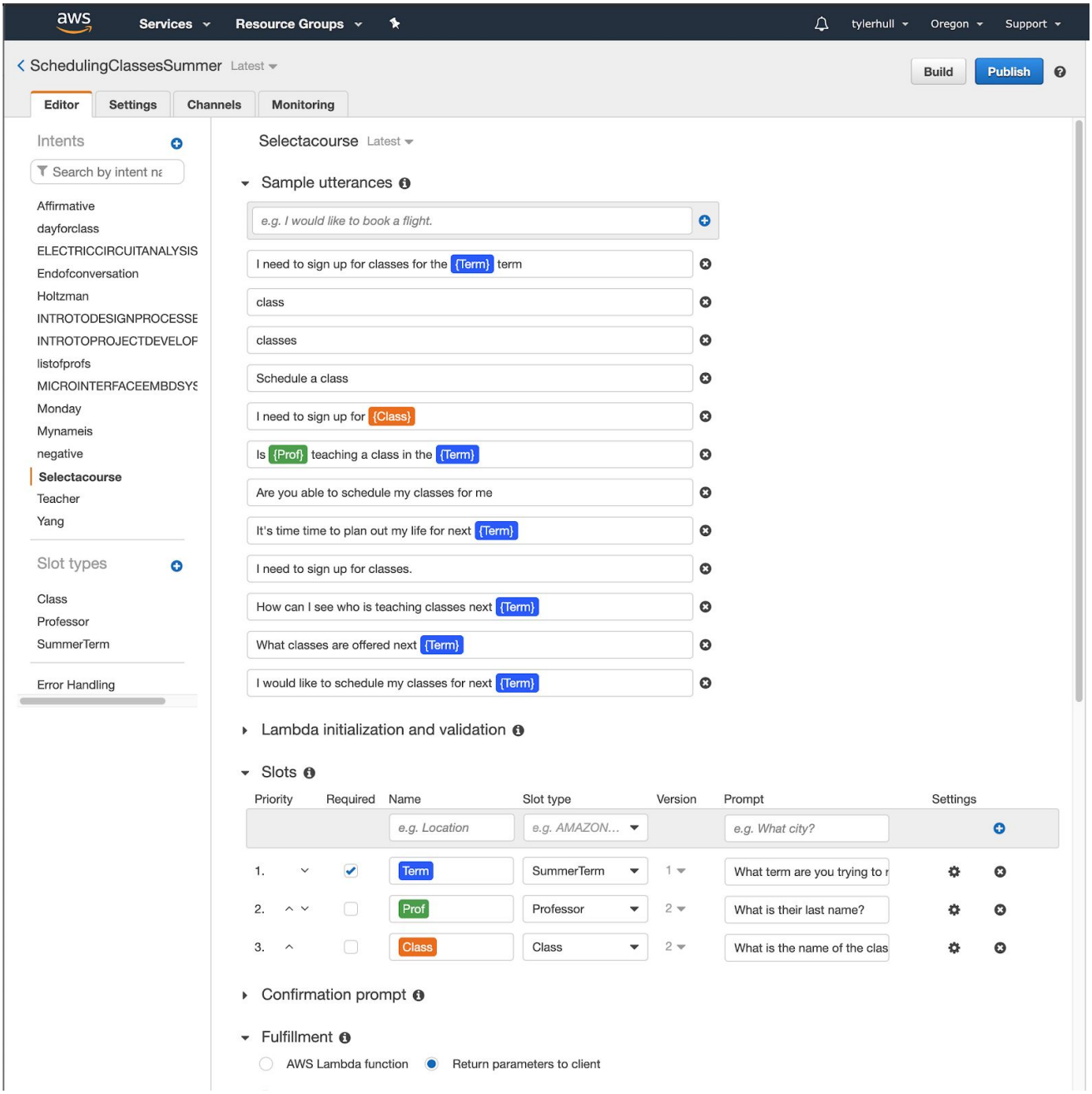


Figure 28: Ella's Amazon LEX Summer Course Conversation buildout.

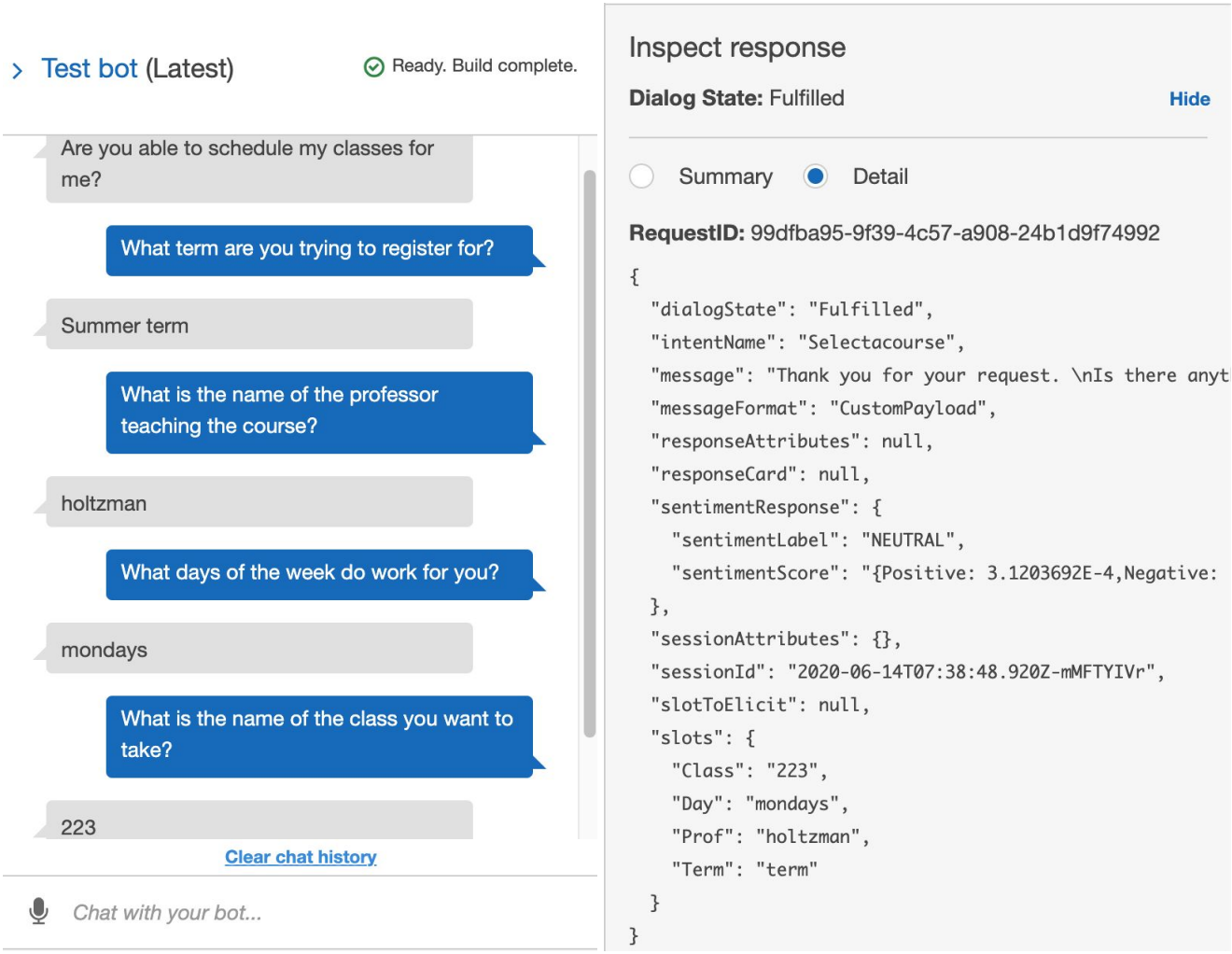


Figure 29: Conversation with Ella through Amazon LEX development platform. Responses are logged and used to improve the model.

When the bot processes your input, it learns about what kinds of things can be said for a certain “intent” and adds those phrases to the model over time. In addition to an audio response, the bot provides a few pieces of identifying information we can use to make the bot more conversational.

For instance, in figure 29 above on the right, you can see that we have a session id. You can also have a user ID, so that “slots” that were filled from a previous conversation can be referenced in future conversations. Also, there is a sentimentLabel and sentimentScore. These attributes are generated when the user writes something that is detected to be angry, happy, etc. This label and score could be used to dynamically respond to a users mood and make talking with the bot a much more engaging experience.

Two Main ways to connect to your Python Program

There are two main ways to connect a python program to AWS LEX. The first is to use the boto3 library and runtime service for LEX. Using this library, we should be able to send a few pieces of information that identify our bot and our account. Then the server would process the request and return the information from the server in a struct. Included in this return data should be an audio file we can play of the response from the bot. This was the planned method for connecting to LEX, but needs some additional time to work out the kinks in the setup.

Use the Runtime Service with Amazon's boto3 API calls

class **LexRuntimeService.Client**

A low-level client representing Amazon Lex Runtime Service:

```
import boto3

client = boto3.client('lex-runtime')
```

Can Send and Receive Audio to Polly through "inputStream"

```
response = client.post_content(
    botName='string',
    botAlias='string',
    userId='string',
    sessionAttributes={...}|[...]|123|123.4|'string'|True|None,
    requestAttributes={...}|[...]|123|123.4|'string'|True|None,
    contentType='string',
    accept='string',
    inputStream=b'bytes'|file
)
```

The next method for interacting with the bot is to use the Command Line Interface. Many AWS account functions can be handled with the CLI and it seems like a very robust tool for working with AWS. I didn't spend a lot of time exploring how this might work, but there is a tutorial on how to use the CLI with LEX. It should be possible to send the required information to the command line from a python program and this would get around the trouble with boto3. Since the CLI should be connected to the AWS account through login credentials, this might be an easier route than working with the library.

Below is an example of a request and the return values from an interaction with LEX using the CLI.

Use the Amazon CLI to Send and Receive Messages

```
aws lex-runtime post-text \
    --region region \
    --bot-name OrderFlowersBot \
    --bot-alias "\$LATEST" \
    --user-id UserOne \
    --input-text "i would like to order flowers"
```

```
{
    "slotToElicit": "FlowerType",
    "slots": {
        "PickupDate": null,
        "PickupTime": null,
        "FlowerType": null
    },
    "dialogState": "ElicitSlot",
    "message": "What type of flowers would you like to order?",
    "intentName": "OrderFlowers"
}
```

Getting Started with Amazon Lex

There are lots of tutorials for getting setup. I used the AWS tutorials and they were really helpful in getting a model going. I like the idea of using this service because the account is free to set up and use up to a certain level of server requests. This limit is very unlikely to be met from a student working on a project like this.

There are a lot of options for ways to connect LEX to other AWS services and that expandability is very powerful. Additionally, it's likely that this service will continue to be supported for some time since it runs on the same backend as one of Amazon's flagship products.

However, it is a little bit complicated to set up the bot for conversations and connecting to other services is very complicated as well. I think it's probably worth the time and effort to continue working on this since experience with AWS is good for the skillset and there are lots of opportunities to expand the work.

A simpler implementation might be to process the responses locally or with a Prolog program and send out the response to Amazon Polly for Text to Speech (TTS).

Conclusion

Challenges and Difficulties

- A major difficulty I faced was selecting a software program for doing animation. I spent a lot of time researching different options and watching tutorials or trying to find anyone doing similar projects. If I had this to do over, I would have just started working on Blender. My simple implementation worked well for beginning this project, but expanding the capabilities of the Avatar really requires some 3D rendering or very complicated groups of 2D art.
- This was my first time working with OpenCV. It took some time to understand getting it installed and the reversed color ordering (It uses BGR instead of RGB). I was able to find a lot of good resources on

this and there are many good books out there for OpenCV. I recommend taking a look at the PACKT Publishing books if you can. They have new titles coming out monthly and they stay pretty up to date on versions of software because of this. Having the wrong versions and packages of different open source software you are trying to piece together is really terrible to deal with and troubleshoot.

- Installing wxpython on my computer was really terrible, but it wasn't entirely wxpython's fault. When using wxpython on linux, you need to build the "wheel" or package of code, yourself. The process for this isn't really documented in a step by step way. But the real problem was, there are no linux drivers for the macbook pro "facetimeHD" webcam. So even if you compiled everything and then tried to test your setup, you would get errors and the camera would never open. I was able to fix this only after installing some 3rd party drivers and installing the camera module, but any updates will break it.
- Getting Amazon LEX to connect to my python program did not go as planned. I wasn't able to get this piece working with the avatar to reproduce speech. I think this is more of a settings and account issue so hopefully other students are able to get this going fairly painlessly. Building out the bot conversation model took a lot of time, but it works really well with the web system once you have it setup. It's very reliable and I tested it with several microphones.

Discussion or project results

Overall, this was a fun project to work on. I enjoyed the artistic aspects of the project as well as exploring OpenCV and AWS platforms for the first time. The emotion recognition for the Avatar is working quite well and has many opportunities for expansion with additional haar cascade classifiers or additional emotions. The phonetic mouth position capture is working fairly well and just needs a few tweaks or to be broken out as its own mode. The program is very close to being able to be connected to AWS LEX for giving the Avatar a conversational interface. I think this would be a very nice addition to the program and I'm sad I couldn't quite get it all connected.

In addition, I think the platform created with this project will help other students get a faster start. Hopefully future students can work on a Blender model and connect LEX or some other language model to the Avatar. Eventually, it will be great to write a program, simulate it with the avatar, then generate or validate code that can run the robot version.

Appendix

Download Files from Github Repository

https://github.com/tylerhull/Avatar

Program Requirements

Make sure to install these requirements to get things running on your system. Put all of these into a .txt file called “requirements.txt” and run the following command to install all the requirements from the doc with pip.

This sometimes works... you can also just install everything manually. You may need different versions based on your setup. Good luck!

pip3 install -r requirements.txt

wxPython==4.0.5
numpy==1.18.1
scipy==1.4.1
matplotlib==3.1.2
requests==2.22.0
opencv-contrib-python==4.2.0.32
opencv-python==4.2.0.32
rawpy==0.14.0
ExifRead==2.1.2
tensorflow==2.0.1
wheels/wxPython-4.0.7.post2-cp38-cp38-linux_x86_64.whl

Program Code

Avatar_emo.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


"""
    An app that detects faces, reads emotions, and displays an avatar with the
    same emotions as the detected face.


    How to run the program:
    1.  run the image collection program with: python3 avatar_emo.py collect
        - take images of your face to use with the training for each emotion
    2.  train the MLP on your images for each emotion or mouth shape
        - do this by running the command: python3 train_classifier.py --data
          <pathToYourData> --save <pathToWhereYouWantToSave>
    3. Run the emotions program with: python3 avatar_emo.py demp --classifier
```

```
        <PathToWhereYouSavedTheClassifier>

"""

import argparse     # Library for parsing arguments from user
import cv2          # OpenCV library needed for working with openCV
import numpy as np  # Num py library to work with arrays
import time         # Library to work with time
import boto3        # This is the library for talking to amazon lex

import wx           # Library for working with the wxPython GUI
from pathlib import Path

from data.store import save_datum, pickle_load  # Used to save and read from files
from data.process import _pca_featurize         # Needed to read face features
from detectors import FaceDetector              # Detects a face in the frame
from wx_gui import BaseLayout                   # Get the base layout template from wx

### GLOBAL VARIABLES ###
lastlabel = 'neutral' # Initialize variable for keeping track of emotion state


### This class runs when DEMO mode is selected ###
class FacialExpressionRecognizerLayout(BaseLayout):
    def __init__(self, *args,
                 clf_path=None,
                 **kwargs):
        super().__init__(*args, **kwargs)
        self.clf = cv2.ml.ANN_MLP_load(str(clf_path / 'mlp.xml'))

        self.index_to_label = pickle_load(clf_path / 'index_to_label')
        self.pca_args = pickle_load(clf_path / 'pca_args')

        self.face_detector = FaceDetector(
            face_cascade='params/haarcascade_frontalface_default.xml',

            # Use this one if you don't have glasses
            eye_cascade='params/haarcascade_lefteye_2splits.xml')

            # Use this one if you have glasses
            #eye_cascade='params/haarcascade_eye_tree_eyeglasses.xml')

    def featurize_head(self, head):
        return _pca_featurize(head[None], *self.pca_args)

    def augment_layout(self):
```

```
        """Initializes GUI"""
        # initialize data structure
        self.samples = []
        self.labels = []

        pnl4 = wx.Panel(self, -1)
        hbox4 = wx.BoxSizer(wx.HORIZONTAL)
        img = cv2.imread('neutral.jpg',1)
        cv2.imshow('AVATAR', img)

        print(type(img))
        # <class 'numpy.ndarray'>

        print(img.shape)
        # (225, 400, 3)



### Send Responses to Amazon LEX ###
        #txt = input("You can talk to Ella by typing something to her.")

        #response = client.post_text(
        #botName='PizzaOrderingBot',
        #botAlias='MeanPizzaBot',
        #userId='user1',
        #sessionAttributes={
    #        'string': 'string'
    #    },
    #    requestAttributes={
    #        'string': 'string'
    #    },
    #    inputText=txt
    #    )


        # arrange all horizontal layouts vertically
        self.panels_vertical.Add(pnl4, flag=wx.EXPAND | wx.BOTTOM, border=1)

    #    pass
    def update(label):
        global lastlabel
        lastlabel = label
        return lastlabel

    def process_frame(self, frame_rgb: np.ndarray) -> np.ndarray:
        success, frame, self.head, (x, y) = self.face_detector.detect_face(
```

```
        frame_rgb)
    if not success:
        return frame

    success, head = self.face_detector.align_head(self.head)
    if not success:
        return frame

    # We have to pass [1 x n] array predict.
    _, output = self.clf.predict(self.featurize_head(head))
    label = self.index_to_label[np.argmax(output)]

    # Draw predicted label above the bounding box.
    cv2.putText(frame, label, (x, y - 20),
                cv2.FONT_HERSHEY_COMPLEX, 1, (0, 255, 0), 2)

    global lastlabel

    # Display the appropriate image for the emotion label
    if lastlabel != label:
        cv2.destroyWindow('img')
        if label == 'happy':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img = cv2.imread('happy.jpg',1)
            cv2.imshow('AVATAR', img)
            lastlabel = label

        elif label == 'sad':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('sad.jpg',1)
            cv2.imshow('AVATAR', img2)

            cv2.waitKey(100)
            img2 = cv2.imread('sad1.jpg',1)
            cv2.imshow('AVATAR', img2)

            cv2.waitKey(100)
            img2 = cv2.imread('sad2.jpg',1)
            cv2.imshow('AVATAR', img2)

            cv2.waitKey(100)
            img2 = cv2.imread('sad3.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'surprised':  #if this was selected, go to collection mode.
```

```
            cv2.waitKey(1)
            img2 = cv2.imread('surprised.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'angry':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('angry.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'disgusted':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('disgusted.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'neutral':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('neutral.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

    ### If the user is running mouth shapes setting, then display those ###
        elif label == 'AH':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('AH.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'D':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('D.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'Ee':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('Ee.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'F':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('F.jpg',1)
            cv2.imshow('AVATAR', img2)
```

```
            lastlabel = label

        elif label == 'L':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('L.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'M':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('M.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'oh':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('oh.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'R':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('R.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'S':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('S.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'Uh':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('Uh.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label

        elif label == 'Woo':  #if this was selected, go to collection mode.
            cv2.waitKey(1)
            img2 = cv2.imread('Woo.jpg',1)
            cv2.imshow('AVATAR', img2)
            lastlabel = label


    return frame
```

```
### This class runs when COLLECTION mode is selected ###
class DataCollectorLayout(BaseLayout):

    def __init__(self, *args,
                 training_data='data/cropped_faces.csv',
                 **kwargs):
        super().__init__(*args, **kwargs)
        self.face_detector = FaceDetector(
            face_cascade='params/haarcascade_frontalface_default.xml',
            eye_cascade='params/haarcascade_lefteye_2splits.xml')

        self.training_data = training_data

    def augment_layout(self):
        """Initializes GUI"""
        # initialize data structure
        self.samples = []
        self.labels = []

        ### create a horizontal layout with all buttons for Emotions ###
        pnl2 = wx.Panel(self, -1)
        self.neutral = wx.RadioButton(pnl2, -1, 'neutral', (10, 10),
                                      style=wx.RB_GROUP)
        self.happy = wx.RadioButton(pnl2, -1, 'happy')
        self.sad = wx.RadioButton(pnl2, -1, 'sad')
        self.surprised = wx.RadioButton(pnl2, -1, 'surprised')
        self.angry = wx.RadioButton(pnl2, -1, 'angry')
        self.disgusted = wx.RadioButton(pnl2, -1, 'disgusted')

        self.Ah = wx.RadioButton(pnl2, -1, 'Ah')

        self.D = wx.RadioButton(pnl2, -1, 'D')
        self.Ee = wx.RadioButton(pnl2, -1, 'Ee')
        self.F = wx.RadioButton(pnl2, -1, 'F')
        self.L = wx.RadioButton(pnl2, -1, 'L')
        self.M = wx.RadioButton(pnl2, -1, 'M')
        self.oh = wx.RadioButton(pnl2, -1, 'oh')
        self.R = wx.RadioButton(pnl2, -1, 'R')
        self.S = wx.RadioButton(pnl2, -1, 'S')
        self.Uh = wx.RadioButton(pnl2, -1, 'Uh')
        self.Woo = wx.RadioButton(pnl2, -1, 'Woo')

        hbox2 = wx.BoxSizer(wx.VERTICAL)
        hbox2.Add(self.neutral, 1)
        hbox2.Add(self.happy, 1)
```

```
        hbox2.Add(self.sad, 1)
        hbox2.Add(self.surprised, 1)
        hbox2.Add(self.angry, 1)
        hbox2.Add(self.disgusted, 1)
        hbox2.Add(self.Ah, 1)
        hbox2.Add(self.D, 1)
        hbox2.Add(self.Ee, 1)
        hbox2.Add(self.F, 1)
        hbox2.Add(self.L, 1)
        hbox2.Add(self.M, 1)
        hbox2.Add(self.oh, 1)
        hbox2.Add(self.R, 1)
        hbox2.Add(self.S, 1)
        hbox2.Add(self.Uh, 1)
        hbox2.Add(self.Woo, 1)
        pnl2.SetSizer(hbox2)

        # create horizontal layout with single snapshot button
        pnl3 = wx.Panel(self, -1)
        self.snapshot = wx.Button(pnl3, -1, 'Click Here to Take Photo')
        self.Bind(wx.EVT_BUTTON, self._on_snapshot, self.snapshot)
        hbox3 = wx.BoxSizer(wx.HORIZONTAL)
        hbox3.Add(self.snapshot, 1)
        pnl3.SetSizer(hbox3)

        """
        ### This section will add a new panel to the layout, but it overides
        ### the emotion data. Looks better in the GUI though.
        ### create a horizontal layout with buttons for mouth shapes ###
        pnl4 = wx.Panel(self, -1)
        self.Ah = wx.RadioButton(pnl4, -1, 'Ah', (10, 10),
                                 style=wx.RB_GROUP)
        self.D = wx.RadioButton(pnl4, -1, 'D')
        self.Ee = wx.RadioButton(pnl4, -1, 'Ee')
        self.F = wx.RadioButton(pnl4, -1, 'F')
        self.L = wx.RadioButton(pnl4, -1, 'L')
        self.M = wx.RadioButton(pnl4, -1, 'M')
        self.oh = wx.RadioButton(pnl4, -1, 'oh')
        self.R = wx.RadioButton(pnl4, -1, 'R')
        self.S = wx.RadioButton(pnl4, -1, 'S')
        self.Uh = wx.RadioButton(pnl4, -1, 'Uh')
        self.Woo = wx.RadioButton(pnl4, -1, 'Woo')

        hbox4 = wx.BoxSizer(wx.HORIZONTAL)
        hbox4.Add(self.Ah, 1)
        hbox4.Add(self.D, 1)
```

```
        hbox4.Add(self.Ee, 1)
        hbox4.Add(self.F, 1)
        hbox4.Add(self.L, 1)
        hbox4.Add(self.M, 1)
        hbox4.Add(self.oh, 1)
        hbox4.Add(self.R, 1)
        hbox4.Add(self.S, 1)
        hbox4.Add(self.Uh, 1)
        hbox4.Add(self.Woo, 1)
        pnl4.SetSizer(hbox4)
        """

        # arrange all horizontal layouts panels vertically
        self.panels_vertical.Add(pnl2, flag=wx.EXPAND | wx.BOTTOM, border=1)
    #   self.panels_vertical.Add(pnl4, flag=wx.EXPAND | wx.BOTTOM, border=1)
        self.panels_vertical.Add(pnl3, flag=wx.EXPAND | wx.BOTTOM, border=1)



    def process_frame(self, frame_rgb: np.ndarray) -> np.ndarray:
        """
        Add a bounding box around the face if a face is detected.
        """
        _, frame, self.head, _ = self.face_detector.detect_face(frame_rgb)
        return frame

    def _on_snapshot(self, evt):
        """Takes a snapshot of the current frame

            This method takes a snapshot of the current frame, preprocesses
            it to extract the head region, and upon success adds the data
            sample to the training set.
        """
        ## Give labels to images for emotions
        if self.neutral.GetValue():
            label = 'neutral'
        elif self.happy.GetValue():
            label = 'happy'
        elif self.sad.GetValue():
            label = 'sad'
        elif self.surprised.GetValue():
            label = 'surprised'
        elif self.angry.GetValue():
            label = 'angry'
        elif self.disgusted.GetValue():
            label = 'disgusted'
```

```
        ## Give labels to images for mouth shapes
        if self.Ah.GetValue():
            label = 'Ah'
        elif self.D.GetValue():
            label = 'D'
        elif self.Ee.GetValue():
            label = 'Ee'
        elif self.F.GetValue():
            label = 'F'
        elif self.L.GetValue():
            label = 'L'
        elif self.M.GetValue():
            label = 'M'
        elif self.oh.GetValue():
            label = 'oh'
        elif self.R.GetValue():
            label = 'R'
        elif self.S.GetValue():
            label = 'S'
        elif self.Uh.GetValue():
            label = 'Uh'
        elif self.Woo.GetValue():
            label = 'Woo'

        if self.head is None:
            print("No face detected")
        else:
            success, aligned_head = self.face_detector.align_head(self.head)
            if success:
                save_datum(self.training_data, label, aligned_head)
                print(f"Saved {label} training datum.")
            else:
                print("Could not align head (eye detection failed?)")

### GUI Layout for Collection MOde ###
def run_layout(layout_cls, **kwargs):
    # open webcam
    capture = cv2.VideoCapture(0)
    # opening the channel ourselves, if it failed to open.
    if not(capture.isOpened()):
        capture.open()

    # Set the size of the openCV window that opens
    capture.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
```

```
    # start graphical user interface
    app = wx.App()
    layout = layout_cls(capture, **kwargs)
    layout.Center()
    layout.Show()
    app.MainLoop()

# GUI Layout for Demo MOde
def run_layout2(layout_cls, **kwargs):
    # open webcam
    capture = cv2.VideoCapture(0)
    # opening the channel ourselves, if it failed to open.
    if not(capture.isOpened()):
        capture.open()

    # Set the size of the openCV window that opens
    capture.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    # start graphical user interface
    app = wx.App()
    layout = layout_cls(capture, **kwargs)
    layout.Center()
    layout.Show()
    app.MainLoop()


# Here we check to see if the user wants to collect new data or if they
# want to run the program with the trained dataset.
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('mode', choices=['collect', 'demo'])
    parser.add_argument('--classifier', type=Path)
    args = parser.parse_args()

    if args.mode == 'collect':  #if this was selected, go to collection mode.
        run_layout(DataCollectorLayout, title='Select Emotion and Take Photo to Collect
Data')
    elif args.mode == 'demo':   #Otherwise, we want to run the program.
        assert args.classifier is not None, 'you have to provide --classifier'
        run_layout2(FacialExpressionRecognizerLayout,
                    title='Facial Expression Recognizer',
                    clf_path=args.classifier)
```

```
    #    run_layout3(FacialExpressionRecognizerLayout,
    #                title='Facial Expression Recognizer',
    #                clf_path=args.classifier)
```

Detectors.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""A module that contains various detectors"""

import cv2
import numpy as np


class FaceDetector:
    """Face Detector

        This class implements a face detection algorithm using a face cascade
        and two eye cascades.
    """

    def __init__(self, *,
                 face_cascade='params/haarcascade_frontalface_default.xml',
                 eye_cascade='params/haarcascade_lefteye_2splits.xml',
                 # Use this one if you have glasses
                 #eye_cascade='params/haarcascade_eye_tree_eyeglasses.xml')
                 scale_factor=4):

        # resize images before detection
        self.scale_factor = scale_factor

        # load pre-trained cascades
        self.face_clf = cv2.CascadeClassifier(face_cascade)
        if self.face_clf.empty():
            raise ValueError(f'Could not load face cascade "{face_cascade}"')
        self.eye_clf = cv2.CascadeClassifier(eye_cascade)
        if self.eye_clf.empty():
            raise ValueError(
                f'Could not load eye cascade "{eye_cascade}"')

    def detect_face(self, rgb_img, *, outline=True):
        """Performs face detection
```

```
        This method detects faces in an RGB input image.
        The method returns True upon success (else False), draws the
        bounding box of the head onto the input image (frame), and
        extracts the head region (head).

        :param frame: RGB input image
        :returns: success, frame, head
    """
    frameCasc = cv2.cvtColor(cv2.resize(rgb_img, (0, 0),
                                        fx=1.0 / self.scale_factor,
                                        fy=1.0 / self.scale_factor),
                             cv2.COLOR_RGB2GRAY)
    faces = self.face_clf.detectMultiScale(
        frameCasc,
        scaleFactor=1.1,
        minNeighbors=3,
        flags=cv2.CASCADE_SCALE_IMAGE) * self.scale_factor

    # if face is found: extract head region from bounding box and get location
    for (x, y, w, h) in faces:
        if outline:
            cv2.rectangle(rgb_img, (x, y), (x + w, y + h), (100, 255, 0),
                          thickness=2)
        head = cv2.cvtColor(rgb_img[y:y + h, x:x + w],
                            cv2.COLOR_RGB2GRAY)
        return True, rgb_img, head, (x, y)

    return False, rgb_img, None, (None, None)

# Find the centers of the eyes so we can align head for consistency
def eye_centers(self, head, *, outline=False):
    height, width = head.shape[:2]

    eyes = self.eye_clf.detectMultiScale(head,
                                         scaleFactor=1.1,
                                         minNeighbors=3,
                                         flags=cv2.CASCADE_SCALE_IMAGE)
    if len(eyes) != 2: #Might be more than one face. Can't detect for 2.
        raise RuntimeError(f'Number of eyes {len(eyes)} != 2')
    eye_centers = []
    for x, y, w, h in eyes:
        # find the center of the detected eye region
        eye_centers.append(np.array([x + w / 2, y + h / 2]))
        if outline:
            cv2.rectangle(head, (x, y), (x + w, y + h), (10, 55, 0),
                          thickness=2)
```

```
        return eye_centers

    def align_head(self, head):
        """Aligns a head region using affine transformations

            This method preprocesses an extracted head region by rotating
            and scaling it so that the face appears centered and up-right.

            The method returns True on success (else False) and the aligned
            head region (head). Possible reasons for failure are that one or
            both eye detectors fail, maybe due to poor lighting conditions.

            :param head: extracted head region
            :returns: success, head
        """

        # we want the eye to be at 25% of the width, and 20% of the height
        # resulting image should be square (desired_img_width,
        # desired_img_height)
        desired_eye_x = 0.25
        desired_eye_y = 0.2
        desired_img_width = desired_img_height = 200

        try:
            eye_centers = self.eye_centers(head)
        except RuntimeError:
            return False, head

        if eye_centers[0][0] < eye_centers[0][1]:
            left_eye, right_eye = eye_centers
        else:
            right_eye, left_eye = eye_centers

        # scale the distance between eyes to desired length
        eye_dist = np.linalg.norm(left_eye - right_eye)
        eyeSizeScale = (1.0 - desired_eye_x * 2) * desired_img_width / eye_dist

        # get rotation matrix for face
        # get center point between the two eyes and calculate angle so we can align
        eye_angle_deg = 180 / np.pi * np.arctan2(right_eye[1] - left_eye[1],
                                                 right_eye[0] - left_eye[0])
        eye_midpoint = (left_eye + right_eye) / 2
        rot_mat = cv2.getRotationMatrix2D(tuple(eye_midpoint), eye_angle_deg,
                                          eyeSizeScale)

        # shift center of the eyes to be centered in the image
```

```
        rot_mat[0, 2] += desired_img_width * 0.5 - eye_midpoint[0]
        rot_mat[1, 2] += desired_eye_y * desired_img_height - eye_midpoint[1]

        # warp perspective to make eyes aligned on horizontal line and scaled
        # to right size so all out images are the same for training
        res = cv2.warpAffine(head, rot_mat, (desired_img_width,
                                             desired_img_width))

        # return success if we could align the face
        return True, res
```

Train_classifier.py

```
import argparse
import cv2
import numpy as np
from pathlib import Path
from collections import Counter
from data.store import load_collected_data
from data.process import train_test_split
from data.process import pca_featurize, _pca_featurize
from data.process import one_hot_encode
from data.store import pickle_dump


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data', required=True)    # Need path to captured data
    parser.add_argument('--save', type=Path)        # For location to save model
    parser.add_argument('--num-components', type=int,   # Can choose how many EigenVecs
to look at
                        default=20)
    args = parser.parse_args()

    data, targets = load_collected_data(args.data)

    train, test = train_test_split(len(data), 0.8)

    # Use PCA to reduce dimentionality in data and speed up algorithm
    x_train, pca_args = pca_featurize(np.array(data)[train],
                                      num_components=args.num_components)

    encoded_targets, index_to_label = one_hot_encode(targets)
```

```
    # Create an MLP using OpenCV built in MLP create function
    last_layer_count = len(encoded_targets[0])
    mlp = cv2.ml.ANN_MLP_create()
    mlp.setLayerSizes(np.array([args.num_components, 10, last_layer_count],
dtype=np.uint8))
    mlp.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP, 0.1)
    mlp.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM)
    mlp.setTermCriteria((cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS, 30, 0.000001
))

    y_train = encoded_targets[train]

    # Train on the data we provided from the image capture
    mlp.train(x_train, cv2.ml.ROW_SAMPLE, y_train)

    # Use PCA to reduce dimentionality in data and speed up algorithm
    x_test = _pca_featurize(np.array(data)[test], *pca_args)
    _, predicted = mlp.predict(x_test)

    y_hat = np.array([index_to_label[np.argmax(y)] for y in predicted])
    y_true = np.array(targets)[test]

    # Print out our accuracy calculated from the numpy array
    print('Your Training Accuracy was:')
    print(sum(y_hat == y_true) / len(y_hat))

    # If we chose to save, then take the .xml file that was generated so
    # we can use it with OpenCV when we run our program
    if args.save:
        x_all, pca_args = pca_featurize(np.array(data),
num_components=args.num_components)
        mlp.train(x_all, cv2.ml.ROW_SAMPLE, encoded_targets)
        args.save.mkdir(exist_ok=True)
        mlp.save(str(args.save / 'mlp.xml'))
        pickle_dump(index_to_label, args.save / 'index_to_label')
        pickle_dump(pca_args, args.save / 'pca_args')
```

Wx_gui.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
A module containing simple GUI layouts using wxPython
```

```
This file is heavily based on the work of Michael Beyeler.
"""

__license__ = "GNU GPL 3.0 or later"

import numpy as np
import wx
import cv2


class BaseLayout(wx.Frame):
    """ Abstract base class for all layouts in the book.

    A custom layout needs to implement the 2 methods below
        - augment_layout
        - process_frame
    """

    def __init__(self,
                 capture: cv2.VideoCapture,
                 title: str = None,
                 parent=None,
                 window_id: int = -1,  # default value
                 fps: int = 10):
        """
        Initialize all necessary parameters and generate a basic GUI layout
        that can then be augmented using `self.augment_layout`.

        :param parent: A wx.Frame parent (often Null). If it is non-Null,
            the frame will be minimized when its parent is minimized and
            restored when it is restored.
        :param window_id: The window identifier.
        :param title: The caption to be displayed on the frame's title bar.
        :param capture: Original video source to get the frames from.
        :param fps: Frames per second at which to display camera feed.
        """
        # Make sure the capture device could be set up
        self.capture = capture
        success, frame = self._acquire_frame()
        if not success:
            print("Could not acquire frame from camera.")
            raise SystemExit()
        self.imgHeight, self.imgWidth = frame.shape[:2]

        super().__init__(parent, window_id, title,
```

```
                        size=(self.imgWidth, self.imgHeight + 20))
        self.fps = fps
        self.bmp = wx.Bitmap.FromBuffer(self.imgWidth, self.imgHeight, frame)

        # set up periodic screen capture
        self.timer = wx.Timer(self)
        self.timer.Start(1000. / self.fps)
        self.Bind(wx.EVT_TIMER, self._on_next_frame)

        # set up video stream
        self.video_pnl = wx.Panel(self, size=(self.imgWidth, self.imgHeight))
        self.video_pnl.SetBackgroundColour(wx.BLACK)
        self.video_pnl.Bind(wx.EVT_PAINT, self._on_paint)

        # display the button layout beneath the video stream
        self.panels_vertical = wx.BoxSizer(wx.VERTICAL)
        self.panels_vertical.Add(self.video_pnl, 1, flag=wx.EXPAND | wx.TOP,
                                 border=1)

        self.augment_layout()

        # round off the layout by expanding and centering
        self.SetMinSize((self.imgWidth, self.imgHeight))
        self.SetSizer(self.panels_vertical)
        self.Centre()

    def augment_layout(self):
        """ Augment custom layout elements to the GUI.

        This method is called in the class constructor, after initializing
        common parameters. Every GUI contains the camera feed in the variable
        `self.video_pnl`. Additional layout elements can be added below
        the camera feed by means of the method `self.panels_vertical.Add`
        """
        raise NotImplementedError()

    def _on_next_frame(self, event):
        """
        Capture a new frame from the capture device,
        send an RGB version to `self.process_frame`, refresh.
        """
        success, frame = self._acquire_frame()
        if success:
            # process current frame
            frame = self.process_frame(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
```

```
            # update buffer and paint (EVT_PAINT triggered by Refresh)
            self.bmp.CopyFromBuffer(frame)
            self.Refresh(eraseBackground=False)

    def _on_paint(self, event):
        """ Draw the camera frame stored in `self.bmp` onto `self.video_pnl`.
        """
        wx.BufferedPaintDC(self.video_pnl).DrawBitmap(self.bmp, 0, 0)

    def _acquire_frame(self) -> (bool, np.ndarray):
        """ Capture a new frame from the input device

        :return: (success, frame)
            Whether acquiring was successful and current frame.
        """
        return self.capture.read()

    def process_frame(self, frame_rgb: np.ndarray) -> np.ndarray:
        """Process the frame of the camera (or other capture device)

        :param frame_rgb: Image to process in rgb format, of shape (H, W, 3)
        :return: Processed image in rgb format, of shape (H, W, 3)
        """
        raise NotImplementedError()
```

Process.py

```
import json
import numpy as np
from typing import Callable
import cv2


def featurize(datum):
    return np.array(datum, dtype=np.float32).flatten()


EMOTIONS = {
    'neutral': 0,
    'surprised': 1,
    'angry': 2,
    'happy': 3,
    'sad': 4,
    'disgusted': 5,
    'Ah': 6,
```

```
    'D': 7,
    'Ee': 8,
    'F': 9,
    'L': 10,
    'M': 11,
    'oh': 12,
    'R': 13,
    'S': 14,
    'Uh': 15,
    'Woo': 16
}

REVERSE_EMOTIONS = {v: k for k, v in EMOTIONS.items()}


def int_encode(label):
    return EMOTIONS[label]


def int_decode(value):
    return REVERSE_EMOTIONS[value]


def one_hot_encode(all_labels) -> (np.ndarray, Callable):
    unique_lebels = list(sorted(set(all_labels)))
    index_to_label = dict(enumerate(unique_lebels))
    label_to_index = {v: k for k, v in index_to_label.items()}

    y = np.zeros((len(all_labels), len(unique_lebels))).astype(np.float32)
    for i, label in enumerate(all_labels):
        y[i, label_to_index[label]] = 1

    return y, index_to_label


def train_test_split(n, train_portion=0.8, seed=None):
    if seed:
        np.random.seed(seed)
    indices = np.arange(n)
    np.random.shuffle(indices)
    N = int(n * train_portion)
    return indices[:N], indices[N:]


def _pca_featurize(data, center, top_vecs):
    return np.array([np.dot(top_vecs, np.array(datum).flatten() - center)
```

```
                    for datum in data]).astype(np.float32)


def pca_featurize(training_data, *, num_components=20):
    x_arr = np.array(training_data).reshape((len(training_data),
-1)).astype(np.float32)
    mean, eigvecs = cv2.PCACompute(x_arr, mean=None)

    # Take only first num_components eigenvectors.
    top_vecs = eigvecs[:num_components]
    center = mean.flatten()

    args = (center, top_vecs)
    return _pca_featurize(training_data, *args), args


if __name__ == '__main__':
    print(train_test_split(10, 0.8))
    from data.store import load_collected_data
    data, targets = load_collected_data('data/cropped_faces.csv')
    X, f = pca_featurize(data)
    print(X.shape)
```

Store.py

```
import csv
import pickle
import json
from enum import IntEnum, auto, unique
import sys
csv.field_size_limit(sys.maxsize)


def load_collected_data(path):
    data, targets = [], []
    with open(path, 'r', newline='') as infile:
        reader = csv.reader(infile)
        for label, sample in reader:
            targets.append(label)
            data.append(json.loads(sample))
    return data, targets


def save_datum(path, label, img):
    with open(path, 'a', newline='') as outfile:
        writer = csv.writer(outfile)
```

```
        writer.writerow([label, img.tolist()])


def pickle_dump(f, path):
    with open(path, 'wb') as outfile:
        return pickle.dump(f, outfile)


def pickle_load(path):
    with open(path, 'rb') as infile:
        return pickle.load(infile)


if __name__ == '__main__':
    td = load_collected_data('data/cropped_faces.csv')
    print([len(x) for x in td])
```