Multimodal Interaction A.Y. 2019/2020

GesturePad: voice and gesture-enabled HTML text editor

Di Mambro Angelo
Giona Emanuele

1 Introduction

GesturePad is a text editor capable of producing HTML or Markdown formatted text files, which allows the usage of multiple input modalities[1] to aid in text formatting. Additionally to mouse and keyboard inputs, GesturePad provides an interface based on voice and gestures.

In particular, the vocal interaction is based on a continuous, dictation-style, speaker-independent speech recognition model, whereas the gesture interaction is based on a semaphoric style.

The layout of the report is as follows: we present the architecture of GesturePad and the semantics used in Section 2; then, the technology stack used, implementation details are described in Section 3; afterwards, our overall approach, possible future development plans are commented in Section 4; finally, installation and execution instructions can be found in Appendix A.

The complete source code is available on GitHub, at the URL https://github.com/emanuelegiona/MI2020.

2 Architecture and Semantics

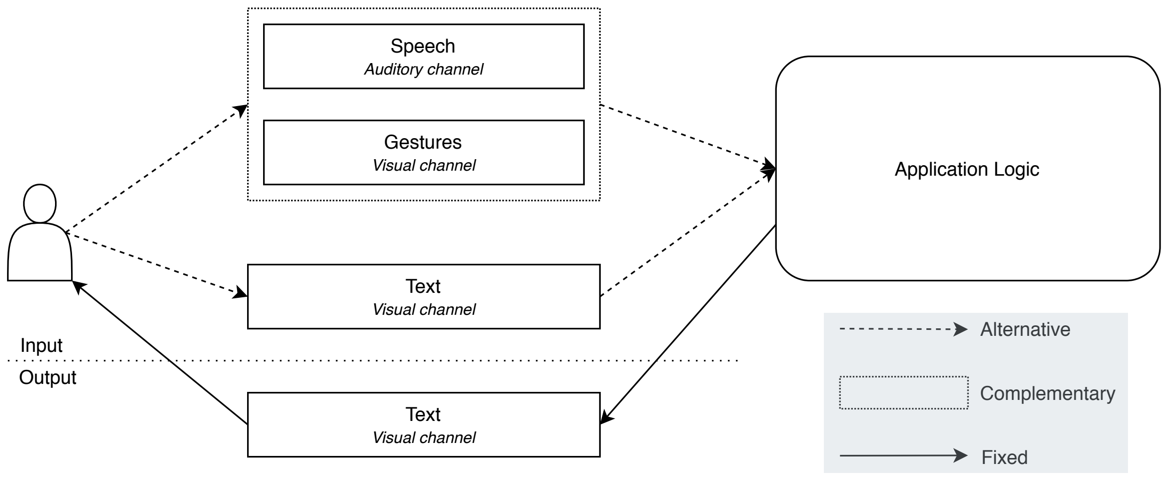


Figure 1: GesturePad interaction model

[1]In this report, for *modality* we intend the physical channel onto which the information is carried (*e.g. visual channel*), whereas for *mode* the way the channel is used to convey information (*e.g. semaphoric gestures*)

The interaction model in GesturePad is depicted in Figure 1, and is easily defined when considering the cooperation modes among input and output modalities[2] separately:

- Input modalities: Textual inputs *alternative* to multimodal inputs. Within the multimodal inputs, speech and gesture modes are *complementary* (non-strict); there are exceptions in case no gestures are provided by the user, in which multimodal inputs boil down to just speech inputs.
- Output modalities: *Specialization* of textual outputs, given the purpose of a text editor.

The choice of these modalities, modes, and cooperation styles stems from the observation that more often than not a user typing a text is forced to stop writing the actual content in order to apply formatting styles. By employing a dictation-style speech recognition model and combining it with semaphoric gestures, we enable the user to simultaneously produce content and apply formatting.

The *alternative* cooperation is simply obtained by imposing a mutual exclusion of the different interaction styles (see Figure 2).

Considering the purpose of a text editor, which requires the visual channel to be specialized to provide the output, the textual interaction can exploit this fact and needs no further pre-processing. This is achieved by directly having the user typing the desired content and/or formatting in the text document to produce.

Instead, a multimodal interaction requires pre-processing and fusion of the two unimodal streams: upon acquiring the speech and gesture media, GesturePad uses their respective recognizers and then applies a fusion at the decision level before manipulating the text document.

In both cases, the user is provided feedback in terms of the effects on the text document, with the option to directly modify the outcome of the interpretations via the standard textual interaction.

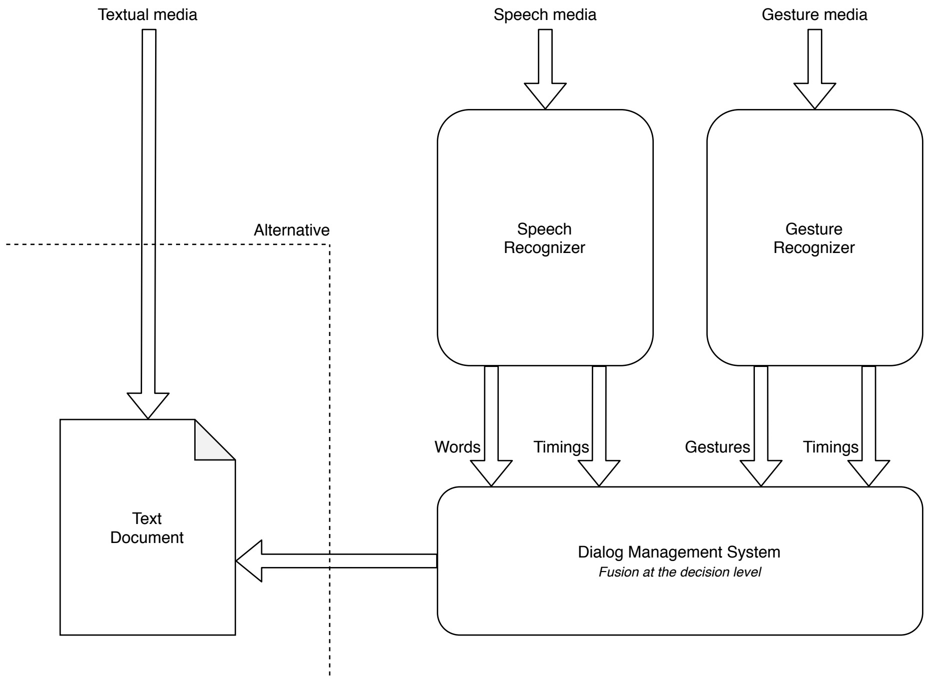


Figure 2: Multimodal flow in GesturePad

[2]From the perspective of the system; *e.g.* an input modality is the channel used by the user to provide information to GesturePad

The decision-level fusion is carried out by making use of an approach similar to the one of *melting pots* with macrotemporal melting strategy. This is due to the fact that utterances from the different modalities may be sequential or non-overlapping, but a temporal window is not needed since there is no strict complementarity: in case the user has to format the content she/he is dictating, gestures are mandatory; on the other hand, in case formatting is not considered, speech and gestures modes can be more appropriately considered in cooperation by *addition*.

This observation allows us to loosen the coordination constraints and not employ a temporal window on utterances. Figure 3 depicts the fusion algorithm.

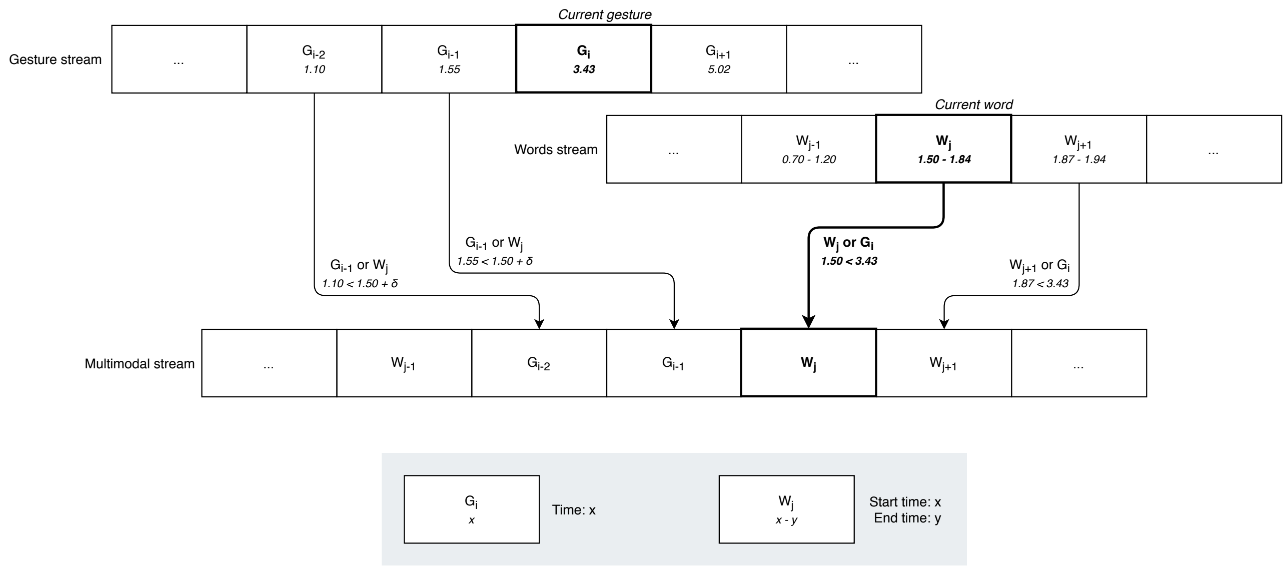


Figure 3: Multimodal fusion in GesturePad

Words and gestures are combined in a multimodal stream after their respective recognition processes, which also provide the timings associated with each individual recognition. These timings are exploited during the multimodal fusion phase in a simplified melting pot strategy. Assuming a gesture G_i with associated timing $T(G_i)$, a word W_j with associated timings $T_s(W_j)$ and $T_e(W_j)$: G_i is considered preceding W_j *iff* its timing is within a certain synchronization tolerance δ from the start of the word:

$$G_i < W_j \iff T(G_i) \leq T_s(W_j) + \delta \tag{1}$$

After a gesture G_i has been added to the multimodal stream, any subsequent word W_j s.t. $T_s(W_j) < T(G_{i+1})$ is added as well (being G_{i+1} the next gesture, if any). The use of δ allows users to pronounce a word they want to be formatted at the same time of signaling the gesture associated to the formatting style desired.

2.1 Semaphoric Gestures

As previously mentioned, GesturePad provides a semaphoric gesture-based interaction to aid in text formatting. Actually it recognizes a set of 11 gestures, showed in Figure 4, that could be further expanded including more commands. Follows a short description about the effect and the correct execution of each gesture in the collection:

- gesture a, bold: once it is correctly recognized all the subsequent text before another such gesture, will be bold. To correctly execute this gesture put the back of the left hand in front of the camera extending the thumb and the pinkie while holding the three middle fingers curled. With the fingers facing forward, the same gesture is the letter Y in the American Manual Alphabet (AMA) [1].

- gesture b, caps lock: once it is correctly recognized all the subsequent text before another such gesture, will be capitalized. To correctly execute this gesture put the palm of the right hand in front of the camera extending the right thumb and index fingers, leaving the other fingers closed to create the letter L.

- gesture c, colon: once it is correctly recognized a colon (:) will be appended at the appropriate position in the text. To correctly execute this gesture put the palm of the right hand in front of the camera extending the right thumb, index and middle fingers, leaving the other fingers closed.

- gesture d, comma: once it is correctly recognized a comma (,) will be appended at the appropriate position in the text. To correctly execute this gesture put the palm of the left hand in front of the camera reproducing the letter C in the AMA.

- gesture e, exclamation mark: once it is correctly recognized an exclamation mark (!) will be appended at the appropriate position in the text. To correctly execute this gesture put the back of the left hand in front of the camera reproducing the letter V in the AMA.

- gesture f, full stop: once it is correctly recognized a full stop will be appended at the appropriate position in the text. To correctly execute this gesture put the palm of the right hand in front of the camera reproducing the digit 5 in the AMA.

- gesture g, italics: once it is correctly recognized all the subsequent text before another such gesture, will be italic. To correctly execute this gesture keep your left hand flatten with the fingers extended and together, put the palm facing to your right keeping the hand in vertical position, and finally skew it a bit (again to your right).

- gesture h, new line: once it is correctly recognized a new line will be started. To correctly execute this gesture keep your right hand flatten with the fingers extended and together and put the palm facing to your left, maintaining the hand in vertical position.

- gesture i, question mark: once it is correctly recognized a question (?) mark will be appended at the appropriate position in the text. To correctly execute this gesture put the back of your left hand in front of the camera and then extend all the fingers bringing all their tips together with the palm facing your eyes[3].

- gesture j, semicolon: once it is correctly recognized a semicolon (;) will be appended at the appropriate position in the text. To correctly execute this gesture show a thumb up with your left hand to the camera.

- gesture k, underlined: once it is correctly recognized all the subsequent text before another such gesture, will be underlined. To correctly execute this gesture keep your left hand flatten with the fingers extended and together and put the palm facing the floor, maintaining the hand in horizontal position.

[3]In Italian, this gesture is called *Mano a borsa* and is commonly used whenever something is not clear

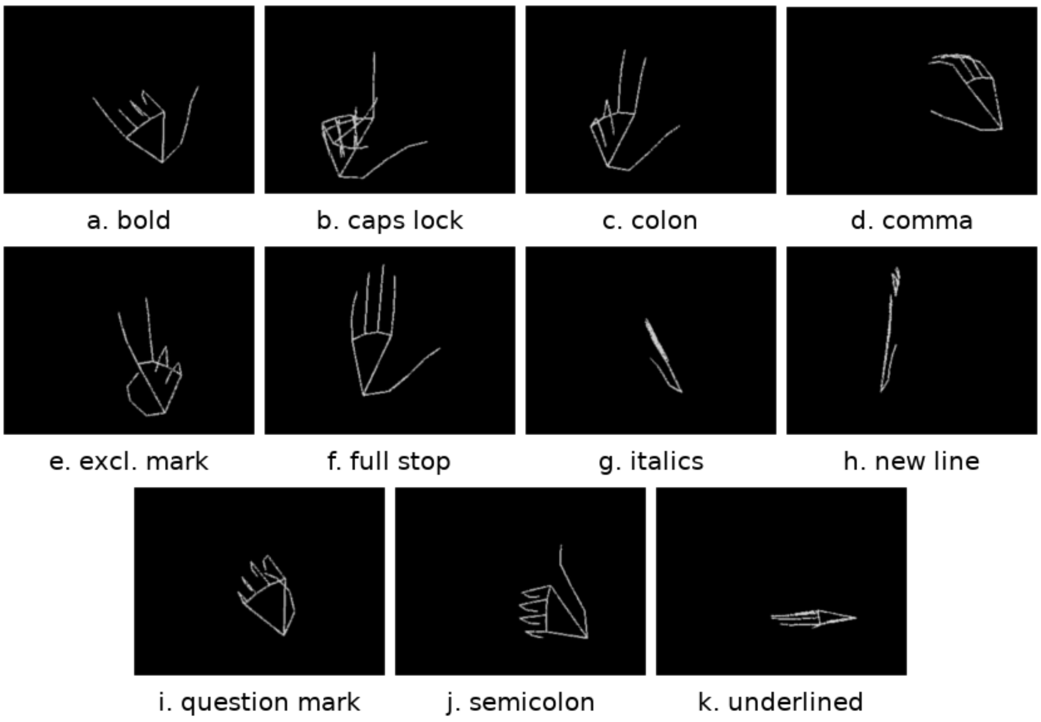


Figure 4: Gestures accepted by GesturePad

3 Implementation Details

The technology stack of GesturePad consists of a Python program driving multiple components, each with its specific purpose: video and audio recording, video pre-processing and stability analysis, speech and gesture recognizers.

The Python program can be logically divided into front end and back end portions, the former containing the GUI definition and the latter the general application logic.

The application logic is therefore implemented in the *backend* Python package, which in turn contains more specialized packages.

3.1 Audio and Video Recording

This component provides the necessary tools used to acquire the native users unimodal streams to interact with the system. The two modalities are captured as:

- Audio streams: the users speech captured from the microphone;
- Video streams: the video that contains gestures frames provided by the users through the webcam.

These streams represent the source input of the system that need to be further processed and refined both offline (audio trimming, gesture frames detection) and online (speech-to-text, gesture recognition).

The following sections describe the contents of the *backend.recording* Python package.

3.1.1 Audio Recording

The Audio module is implemented in the *audio.py* class and contains a group of tools for audio file recording and editing. In particular it provides a set of methods to:

- start and stop the recording from the local microphone of a new audio file, specifying the sample rate at which the audio should be recorded and also the desired duration in seconds. These operations are implemented using the methods inside the *sounddevice* [10] Python library;

- read an already existing audio file from the FS, or write a new recorded file to the FS. These input/output operations are implemented exploiting the methods inside the *soundfile* [11] Python library;

- trim and audio file cutting some parts of it, for example to delete an initial recording noise. These audio editing functions are implemented exploiting the methods inside the *pydub* [8] Python library.

3.1.2 Video Recording

The Video module is implemented in the *video.py* class and contains a group of tools for video file recording. In particular it provides methods to start and stop a new video from the default local acquisition device. Through a real-time pop-up window the user can see what is in recording, frame by frame. Several parameters can be chosen: how many frames-per-second should be acquired, the acquisition resolution and the local path to store the video. These functions are implemented exploiting the methods inside *OpenCV* [6], *PIL* [7], and *Tkinter* [12] Python libraries.

3.2 Audio/Video Pre-processing and Stability Analysis

This component represents the intermediate step before the fusion phase in which the source modalities inputs are processed and refined. In particular it provides a set of methods to:

- trim an audio file, cutting the initial seconds of silence. This silence is present because the audio registration starts almost immediately while the video recording feature from OpenCV has a non-negligible delay before starting. This action is performed to align the two recordings with respect to their duration in order to achieve a better level of synchronization;

- pre-process the video using a customization of *Google MediaPipe* [4] hand-tracking algorithms. Given the video input stream, the video is processed adding an overlay representing the tracked landmarks on the hands: such overlay is structured akin to a graph, in which nodes are pre-determined in the framework, whereas edges simply connect neighboring nodes[4] for a better visualization of the tracked hand;

- analyze a pre-processed video from the previous step, considering a fixed size frame buffer that is *sliden* over time. The frame buffer is used to determine stable frames, which presumably contain a gesture.

The following sections describe the contents of the *backend.mediapipe* Python package.

[4]Neighboring nodes are defined after natural features of human hands: *i.e.* nodes for palm definition, nodes on fingers to detect different positions

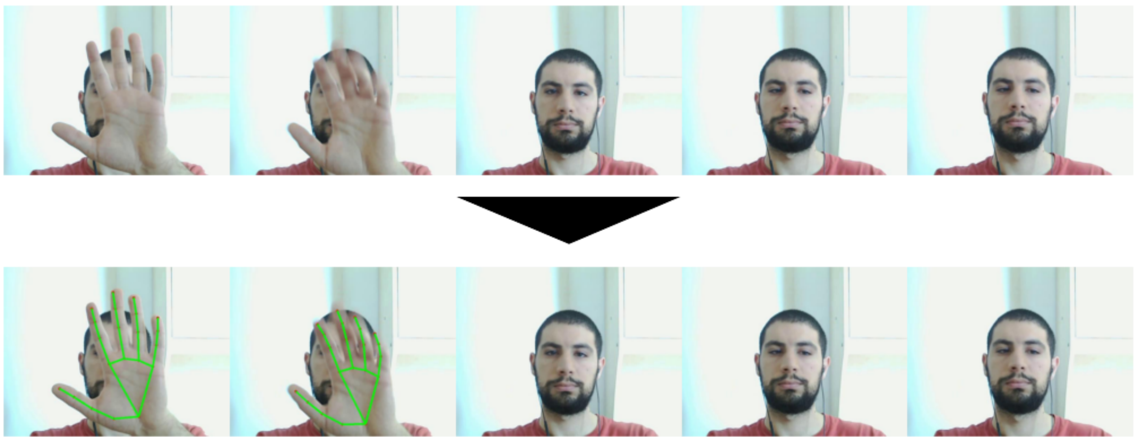


Figure 5: The result of applying Google MediaPipe in the task of hand tracking

3.2.1 Google MediaPipe

A Python module which serves as a wrapper for Google MediaPipe is defined in the file *mediapipe_helper.py*. The source file contains a simple class which handles the execution of an external process to run Google MediaPipe on the original video. Indeed, it provides methods to compile a MediaPipe graph and execute it using the Python operating system interface [9].

Google MediaPipe is a cross-platform computer vision framework based on machine learning, which is written in C++. It defines a high-level language based on graphs in order to ease the development of solutions requiring these features. For our purpose, some original files (namely *demo_run_graph.cc*, *end_loop_calculator.h*, and *landmarks_to_render_data_calculator.cc*) have been replaced with customized ones imported from a similar project [5], publicly available on GitHub. Moreover, files *multi_hand_renderer_cpu.pbtxt* and *multi_hand_tracking_desktop_live.pbtxt* are original modifications of the ones provided directly by Google.

The aforementioned files contain graph definitions to perform the multi-hand tracking task, and the modifications we applied are limited to remove bounding boxes for hands and palms. This stems from the observation that relatively to the purpose of our semaphoric gestures, and more specifically due to our implementation of the gesture recognizer (see Sections 3.2.2, 3.3.2), such boxes have been deemed sources of noise.

3.2.2 Gesture identifier

A Python module to analyze videos via a fixed-size frame buffer w.r.t. the concept of *frame stability*. In a video, for frame stability we intend the effect produced by having a number of subsequent frames that are sufficiently similar to each other.

In order to detect stable frames, we designed an algorithm that iterates over all the frames of a video using a frame buffer. The use of Google MediaPipe allows to easily capture the landmarks drawn on hands while making the rest of pixels black. This is done by creating a mask selecting only pixels in a frame that match the colors associated by MediaPipe to landmarks[5]; then, the mask is reversed and applied to the frame in order to assign the black color to all the non-landmark pixels (see Figure 6). Optionally, a histogram matching algorithm can be executed before the masking operation in order to balance a frame w.r.t. its predecessor in terms of light compensation or avoiding unstable colors due to noisy webcam sensors.

[5]We stick to the original Google MediaPipe colors which are red for nodes and green for edges

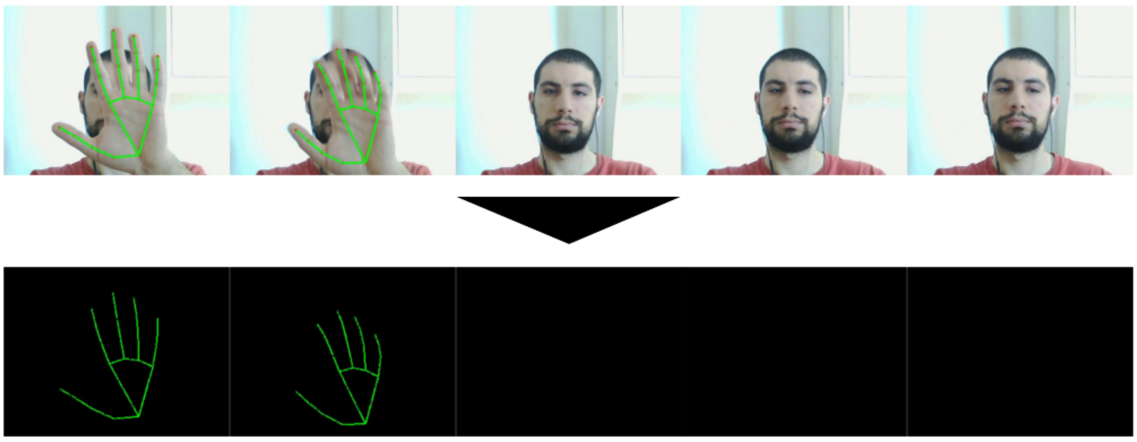


Figure 6: The result of applying our frame enhancement algorithm

Once the frames have been filtered to only obtain a potential gesture, they are normalized and can be added to the frame buffer in order to perform stability analysis. We implement two stability algorithms, and experiment their effects in terms of the frames identified as stable, namely L_n *distance-based similarity* and *Structural similarity*. Given a set F of B frames, the L_n *distance-based similarity* is defined as the average L_n distances of each frame from the mean frame μ_F:

$$stability_{dist}(F) = \frac{1}{B}\sum_{i=1}^{B} |Fi - \mu_F|^n \tag{2}$$

with $|\cdot|$ representing the absolute value operation, and n an arbitrary integer value. Due to the fact that a distance measure is used, a frame F_k is deemed stable in the event $stability_{dist}(F) \leq T_{dist}$ (a threshold on maximum instability tolerated). The best frame F_k is selected as the one minimizing its L_n distance from μ_F.

Instead, the *Structural similarity* stability is defined as the average of pairwise structural similarity values:

$$stability_{sim}(F) = \frac{1}{B}\sum_{i=1}^{B-1} SSIM(F_i, F_{i+1}) \tag{3}$$

with $SSIM(x, y)$ being the value of structural similarity between frames x and y. In this case, a similarity measure is used, therefore a frame F_k is deemed stable in the event $stability_{sim} \geq T_{sim}$ (a threshold on minimum similarity required). The best frame F_k is selected as the one maximizing the similarity to its preceding frame.

Due to the frame enhancement process prior to the stability analysis, we observed a number of frames being identified as stable which did not contain any gesture of ours, but not even pixels tagged with Google MediaPipe landmarks (*i.e.* last 3 frames in Figure 6). This is reasonable given the fact that whenever a frame does not contain any landmark, every pixel in it is black, and a frame buffer containing only black frames is often perceived as stable by either stability algorithm. In order to avoid this kind of false positives, we apply a threshold on the maximum percentage of black pixels in a frame that is detected as being stable: if a frame contains too many black pixels, we can assume there is no landmark and it is a false positive.

A further observation is that users may have different needs and gesture execution times, therefore a user who performs a slower, more still gesture might trigger the frame stability analysis twice in a row and obtain a duplicate gesture. We allow tuning for both the size

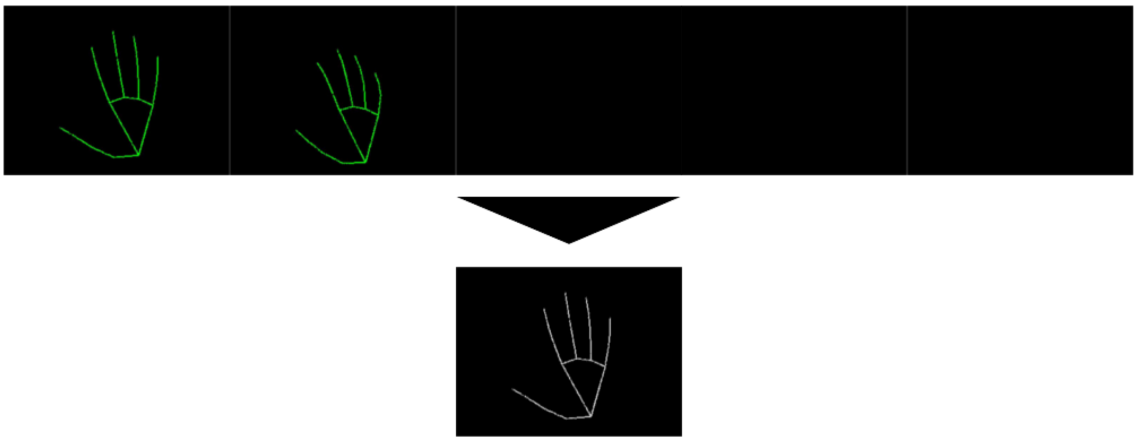


Figure 7: The result of applying our frame stability algorithm

of the frame buffer and the *step*[6] between two frame buffers. However, this might still not be enough to avoid the detection of duplicate stable frames, therefore we implemented an additional similarity comparison between the currently detected stable frame and the previous one. If such similarity exceeds a given threshold, the duplicate stable frame is discarded. An example of stable frame is shown in Figure 7.

At the end of the gesture identification phase, the best frame is returned and associated with the timestamp of the oldest frame in the buffer. This stems from the observation that in order to fill the frame buffer prior to the stability analysis, the oldest frame in the buffer is probably the start of the gesture, after intermediate movements to form such gesture have passed.

3.3 Speech and Gesture Recognizers

This component provides the implementation of the final steps before the multimodal fusion phase. Each unimodal stream is processed by its respective recognizer; for GesturePad, we chose services offered by Google Cloud, namely Cloud Speech-to-Text [2] and Cloud Vision AutoML [3], in order to implement the speech and gesture recognizer respectively. The entire internal composition of the two recognizers is depicted in Figure 8.

The following sections describe the contents of the *backend.clients* Python package.

3.3.1 Speech Recognizer

Google Cloud Speech-to-Text provides an off-the-shelf service that allows to recognize words given an audio file. While there is a convenient REST API, a Python package is also available, and our *speech.py* module provides a wrapper for it in order to comply with the requirements of GesturePad.

Audio files are processed in an asynchronous fashion, by uploading them to the service and keeping a reference to a future result. Once the user wants to retrieve the results, the previously obtained reference can be polled or waited until completion to return the recognized words associated with two timestamps: word recognition start and end, respectively.

[6]The minimum number of frames to skip between two buffers being full: *e.g.* when this parameter is equal to 3, after a frame buffer has been processed the 3 oldest frames in it are discarded, having a distancing effect between frames in two subsequent buffers

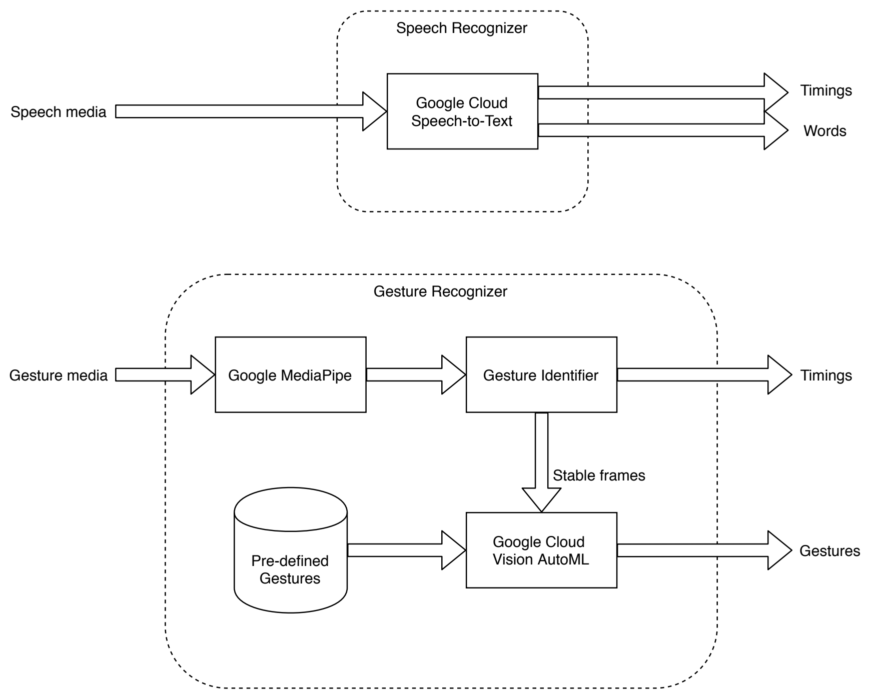


Figure 8: Overall implementation of speech and gesture recognizers previously shown in Figure 2

3.3.2 Gesture Recognizer

Although Google Cloud provides off-the-shelf services for image classification, object detection, and more, for the purpose of GesturePad it is highly unlikely that the same set of gestures of ours had been used before.

Google Cloud Vision AutoML provides a service for building a customized neural network model for image classification, training it, and deploying it to the cloud in an almost completely automatic way. The user only has to provide a labelled dataset, while AutoML will perform all the necessary steps to ensure a quality model is built and trained, comprehending the eventual cloud deployment in order to allow querying the model for predictions on unseen data.

Among the available options, gestures in our application can be identified using a single-label image classification task. In order to have AutoML to work properly, we have built a dataset by defining gestures as labels and associating multiple examples to each label[7], as depicted in Figure 9.

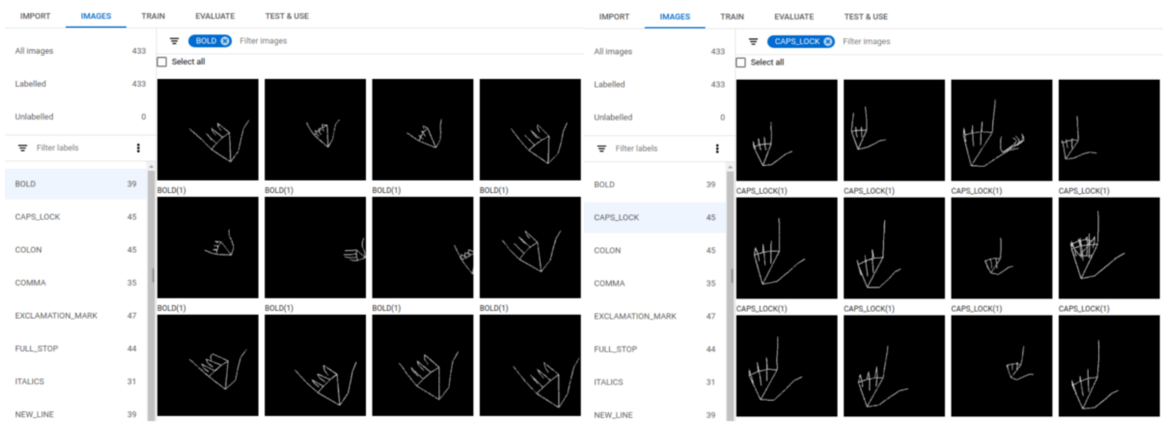
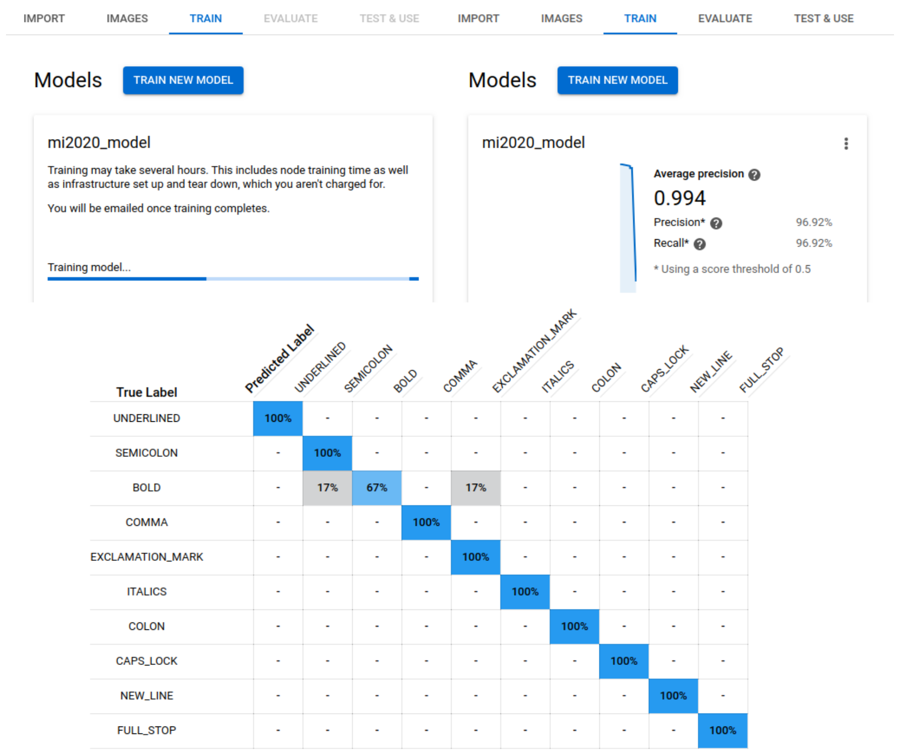


Figure 9: Examples for *bold* and *caps lock* gestures in AutoML (see Section 2.1)

[7]Each gesture has been associated with around 40 examples



mi2020_model

Training may take several hours. This includes node training time as well as infrastructure set up and tear down, which you aren't charged for.

You will be emailed once training completes.

Training model...

mi2020_model

Average precision 0.994

Precision*	96.92%
Recall*	96.92%

* Using a score threshold of 0.5

True Label \ Predicted Label	UNDERLINED	SEMICOLON	BOLD	COMMA	EXCLAMATION_MARK	ITALICS	COLON	CAPS_LOCK	NEW_LINE	FULL_STOP
UNDERLINED	100%	-	-	-	-	-	-	-	-	-
SEMICOLON	-	100%	-	-	-	-	-	-	-	-
BOLD	-	17%	67%	-	17%	-	-	-	-	-
COMMA	-	-	-	100%	-	-	-	-	-	-
EXCLAMATION_MARK	-	-	-	-	100%	-	-	-	-	-
ITALICS	-	-	-	-	-	100%	-	-	-	-
COLON	-	-	-	-	-	-	100%	-	-	-
CAPS_LOCK	-	-	-	-	-	-	-	100%	-	-
NEW_LINE	-	-	-	-	-	-	-	-	100%	-
FULL_STOP	-	-	-	-	-	-	-	-	-	100%

Figure 10: AutoML model training (upper left, upper right) and evaluation (lower center)

After a dataset has been imported in AutoML, a training phase can be started by simply clicking a button in a web browser (see Figure 10), in which a model will be automatically built and trained using the Neural Architecture Search paradigm.

At the end of the training phase, the model can be executed locally by downloading it, or deployed to the cloud and executed through a REST API or Google Cloud library. For the purpose of our project, we chose the latter option: the *gestures.py* Python module defines a wrapper for the Python library that allows to interact with the cloud-deployed custom model. Stable frames are processed in a synchronous fashion, by uploading to the service and waiting for the associated prediction singularly.

Timing information computed during the stability analysis (see Section 3.2.2) is added once all the stable frames have been processed by Cloud AutoML.

3.4 Multimodal Fusion and Formatting Rules

These components represent the conclusive steps in the multimodal flow for GesturePad. As previously described in Section 2, we chose a decision-level fusion for this application. This technique entails executing the recognizers separately on each unimodal stream, and then producing the overall multimodal stream which best fits the application logic.

The following sections describe the contents of *backend.fusion* and *backend.export* Python packages, respectively.

3.4.1 Multimodal Fusion

Our multimodal fuser is defined in the *multimodal_fuser.py* Python module which, given two streams of recognized words and gestures attached with their own timing information, computes an ordering among all the utterances. Due to the appropriate pre-processing step for audio and video files (see Section 3.2), the two streams are somewhat aligned w.r.t. their length, therefore the rather simple mechanism illustrated in Figure 3 is able to produce acceptable results.

A δ parameter (*synchronization tolerance*) can be customized to best fit the user's speaking and gesturing style: indeed, given a gesture G_i and a word W_j with timings $T(G_i)$, and $T_s(W_j)$, $T_e(W_j)$, the gesture is still considered to precede the word even though $T_s(W_j) \leq T(G_i) \leq T_s(W_j) + \delta$.

The rationale behind the customizability of this parameter stems from observing that novice users may have the habit to stop speaking while signaling a gesture, whereas users who are more expert with GesturePad may provide two partially overlapping inputs while intending to apply the gesture before the word.

3.4.2 Formatting Rules

The *format.py* Python module is one enforcing special formatting rules based on the target text format to produce. While GesturePad is mainly focused at writing HTML text, we also implemented a Markdown formatter completely equivalent to the HTML one.

In general, any formatter defined to be used within GesturePad is based on the common abstraction of gestures representing a specific formatting style. For instance, an HTML formatter has to take care of closing tags for rules that work in pairs, whereas a Markdown formatter must account for the lack of underlined formatting style in the target text format.

The implementation of a formatter is rather simple, and within Python a dictionary with gestures as keys and their corresponding strings as values is enough to perform this task.

4 Conclusions and Future Work

We presented GesturePad, a multimodal HTML text editor exploiting auditory (in continuous, dictation-style speech mode) and visual (in semaphoric gestures mode) modalities as input modalities, additionally to the textual mode input.

4.1 Difficulties during Development

OpenCV In the data acquisition phase, and more particularly the video recording step, OpenCV has proved to be both a blessing and a limitation.

While quickly providing a working prototype, its transition into a polished product is hindered by a vaguely defined interface, with too many parameters being set after crowdsourcing instead of reading a carefully written documentation. The main issues are associated with showing the webcam in real-time, which perfectly works in the event of a single recording, but hangs whenever it is performed multiple times. In particular it seems that OpenCV doesn't properly support multi-threading, giving unexpected freezes if the real-time webcam window launcher function is called more than once during the same execution.

We were only able to solve this by totally refraining from using this specific preview functionality from OpenCV and transitioning to Tkinter (which is the framework taking care of the rest of the GUI).

Google MediaPipe Despite its extreme ease of use, Google MediaPipe is the main performance bottleneck in GesturePad, both in terms of application responsiveness and gesture detection accuracy.

Even after having lowered the FPS and resolution parameters of our recorded videos, the landmark computation and drawing task which helps us to gather a distinct hand tracking is the slowest in our application. Regarding the gesture identification accuracy, we observed that

MediaPipe sometimes adds some local changes in the landmark positions even with sufficiently still hands.

We countered this effect by increasing the frame buffer and the inter-gesture *step* sizes in the stability analysis step.

Gesture Identification We encountered several problems regarding this crucial step in GesturePad. First of all, light changes and the presence of webcam sensor noise in the video have challenged intuitive approaches to extract only the landmarks in frames. The original frame stability algorithm was based on a mean squared error (MSE) value under a certain threshold which showed promising results, but was still affected by the aforementioned issues.

Remaining in the RGB space, we implemented histogram matching as a frame pre-processing routine in order to balance colors frame by frame, but it proved to be ineffective and inconvenient in terms of performance. Afterwards, we also implemented alternative stability metrics based on general L_n distances[8] or structural similarity, as widely discussed in Section 3.2.2. We then realized that the main issue was in the frame enhancement process, the most affected by webcam sensor noise, and propagated the error deeper in the gesture identification process.

As a result of this, instead of operating in the RGB space, after converting to the HSV space we create a mask for all green pixels in the range $[[60, 220, 20], [65, 255, 255]]$, which allowed us to use an L_1 distance[9] as stability metric after a normalization to black/white of the enhanced frame.

Google Cloud Vision AutoML The only drawback encountered with AutoML is the impossibility of using a batch prediction service in an interactive application the likes of GesturePad.

This is due to the fact that any batch prediction (even in the extreme case of a single prediction in the batch) takes a *minimum* of 30 minutes before providing results[10].

Our solution consisted in performing a frame-by-frame synchronous prediction.

4.2 Future Work

The general feel when using GesturePad is held back by the limited real-time operation the underlying libraries imply.

At the moment Google MediaPipe is executed in offline mode for synchronization purposes, but an online, real-time operation is supported by the framework. We deliberately chose to operate in offline mode given the fact that we cannot assume the presence of a dedicated GPU to offload the computational load to.

Moreover, while Cloud Vision AutoML operating in a frame-by-frame prediction setting and Cloud Speech-to-Text allowing streaming recognitions[11] would already be able to support a real-time operation, we cannot assume the availability of a large-bandwidth and low-latency network connection which makes the real-time operation actually feasible. An option to reduce the network load would be designing a local gesture detection neural network model, but the same constraint of a real-time Google MediaPipe execution applies (lack of dedicated GPU).

Support for more textual formats and, most importantly, more formatting styles would be appreciated features in GesturePad.

[8] MSE is equivalent to an L_2 distance

[9] L_1 distance is equivalent to Mean Absolute Error (MAE)

[10] Google Cloud Vision AutoML documentation

[11] Google Cloud Speech-to-Text documentation: streaming recognizer

A Installation Instructions

GesturePad has been developed on Ubuntu 18.04 (LTS) with Python 3.6+. See further installation requirements for Google MediaPipe [4].

In order to run this project, a Makefile has been set up to contain all the required libraries and Python packages; for this reason the suggested routine for running this project is the following:

1. Download (and unzip) or clone the project;
2. Move into the main directory (containing the Makefile);
3. Run the `make` command which will: (a) install required system libraries, (b) install the required Python packages, (c) clone Google MediaPipe from its official GitHub repository, and (d) patch the MediaPipe installation with our custom files;
4. Modify the `config.json` file according to your Google Cloud Platform subscription and AutoML settings;
5. Run *gesture_pad.py* in the main project directory and follow the instructions in the GUI.

Note It is advised to set up a Python virtual environment and to download/clone the project into a directory whose parent is not a root-protected directory.

References

[1] *American Manual Alphabet.* URL: https://en.wikipedia.org/wiki/American_manual_alphabet.

[2] Google. *Cloud Speech-to-Text.* URL: https://cloud.google.com/speech-to-text.

[3] Google. *Cloud Vision AutoML.* URL: https://cloud.google.com/automl.

[4] Google. *MediaPipe: Cross-platform ML solutions made simple.* URL: https://mediapipe.dev/.

[5] Anna Kim. *Sign language recognition with RNN and Mediapipe.* 2020. URL: https://github.com/rabBit64/Sign-language-recognition-with-RNN-and-Mediapipe.

[6] OpenCV. *The OpenCV Library.* URL: https://opencv.org/.

[7] PIL. *Python Image Processing.* URL: https://python-pillow.org/.

[8] pydub. *Python package for modifying audio files.* URL: https://pypi.org/project/pydub/.

[9] Python. *Python OS interface.* URL: https://docs.python.org/3/library/os.html#os.system.

[10] sounddevice. *Python package for recording audio files.* URL: https://python-sounddevice.readthedocs.io/en/0.3.15/.

[11] SoundFile. *Python package for writing audio files to disk.* URL: https://pypi.org/project/SoundFile/.

[12] Tkinter. *Python interface to Tcl/Tk.* URL: https://docs.python.org/3/library/tkinter.html.