VIDEO SUBTITLES AS SPEECH BUBBLES

INF573 Project

December 15^{th}, 2021

Astrid Nilsson
Josselin Somerville

CONTENTS

1 INTRODUCTION

Subtitles in series and movies are usually represented as plain and boring text at the bottom of the screen. While it is quite convenient, they are not very fun to look at. Meanwhile, speech bubbles in comics are way more interesting. Given a video and a file with its subtitles, our problem thus consists in displaying these subtitles as speech bubbles on the screen as if the frames were part of a comic strip.

The point of speech bubbles is that they are not in a fixed position. Their position should depend on who is talking and on the environment. Thus, placement of the bubble on the screen should follow certain rules, notably:

- the tail of the bubble should point to the mouth of the person speaking
- the main part of the bubble should occult as little significant objects as possible

In terms of computer vision, this means we are going to need face and mouth recognition algorithms. Face recognition lets us recognize the characters in order to attribute a line to the right person and to avoid placing a bubble on a face. Then, mouth recognition helps us attach the tail of the bubble onto the mouth of the character currently speaking.

To make the resulting video as nice as possible, we can take other problems into account:

- bubbles should not move much between frames, in order to make the text easily readable
- every time a cut occurs, the bubble should disappear or be moved accordingly
- the proportions of the bubble should feel natural (typically, if a line is too long, it should be split into several lines).

2
WORK INSPIRATION

As explained in the introduction, this project requires the assembling of multiple building blocks. Here are all the steps that our algorithm needs to tackle:

1. Read the video and subtitles files
2. Detect face bounding boxes
3. Identify each face to assign the subtitle to the right face
4. Track faces so that the tail of a bubble follows the mouth
5. Detect the position of the mouth given an image of the face
6. Detect if a character is speaking (will be useful in cases where the algorithm does not find the character that is speaking)
7. Detect cuts (to refresh the bubbles)
8. Find the optimal place and size of the speech bubble
9. Draw the bubble
10. Handle cases where the speaker is not on the scene or not found
11. Display the modified frame

And since we want to be able to watch the video, **the processing and displaying of each frame must take less than** $1/fps$**.**

In order to not reinvent the wheel, for most computer vision tasks we used already implemented and trained algorithms. We will present them below:

2.1 Mediapipe

Source: `https://pypi.org/project/mediapipe/`

Mediapipe is a computer vision framework providing API in many languages including Python. Not only does it have amazing precision, it is super fast and that is why we chose this framework.

With Mediapipe's landmarks detection, we were able to solve issues (2) and (5). In only a few milliseconds, Mediapipe returns landmarks which we used to get face boxes and mouth positions.

Figure 1: Example of landmarks to recreate face meshes

2.2 Face recognition

Source: `https://pypi.org/project/face-recognition/`

To identify each face and solve the task (3), we used the *face recognition* Python library. To recognize a face, first the image is fed to an embedder returning a vector of features, then the cosine similarity is computed between the feature vector and the embeddings of reference images. This is obviously less precise than if we had trained a classifier on specific characters; however, it works for anyone without training. Therefore our algorithm works with every single TV show.

As we will see later in this paper, one problem was the accuracy of this face recognition method. If our method ever gets democratized, a classifier would be trained only for characters present in the TV show, subsequently yielding better results.

2.3 PySceneDetect

Source: `https://pyscenedetect.readthedocs.io/en/latest/`

This library helps us determine the timestamps for which cuts occur in the video, solving issue (7). This was not used in the final version of the algorithm but it would be a nice upgrade, solving issues discovered in the results section.

3
OUR METHOD

3.1 Tracking and identifying each face and mouth

3.1.1 • tracking the faces

Even though **MediaPipe** provided us with landmarks for each face, there was still quite a lot of work to do. First, each face must be identified. As explained, this was done using the **face recognition** library. However, recognizing a face requires to embed it into a vector of features which is extremely costly (around 150ms on my computer which has an Intel i7 processor and Nvidia GTX1650 graphics card). Therefore, it is not possible to recognize each face at each frame.

Instead, our approach was based on tracking: if a face was detected close to a face in the previous frame, then we consider that it is the same face. Therefore if the face had already been identified, we do not need to identify it again since it is the same person. In order to improve the results, instead of comparing to the previous position of a face, we predicted the position of a previous face. To do so, we used a Taylor approximation. Here is the interpolation formula *(with p_i the position of the face in the frame i, $\hat{p_i}$ the predicted position, t_i the time of the frame i, and $d_{frame} = t_n - t_{n-1}$)*:

$$\hat{p_n} = p_{n-1} + d_{frame} * \frac{p_{n-1} - p_{n-2}}{t_{n-1} - t_{n-2}} + d_{frame}^2 * \frac{\frac{p_{n-1}-p_{n-2}}{t_{n-1}-t_{n-2}} - \frac{p_{n-2}-p_{n-3}}{t_{n-2}-t_{n-3}}}{t_{n-1} - t_{n-2}}$$

Then we consider that two faces j and k are the same if $||\hat{p_n}[j] - p_n[k]|| \leq \epsilon$, with ϵ a threshold.

In reality, without considering the size of the faces, this does not work well since we don't want a small face to be considered the same one as a huge face in the previous frame. Therefore, just like the position, we interpolate the width w_n and height h_n of each face.

Finally, we compute the similarity between a face k detectected in the frame $n-1$ and a face k in the frame n :

$$similarity[j,k] = \frac{\hat{w_n}[j] * \hat{h_n}[j]}{w_n[k] * h_n[k]} * \left(1 - ||\hat{p_n}[j] - p_n[k]||^{0.4}\right)$$

Therefore if we want to assign detected faces to previously detected faces, we build a similarity matrix S of size a x b with a the number of faces detected in the frame n and b the number of faces detected in the frame $n-1$. The matrix is computed as follows : $S_{j,k} = similarity[j,k]$.

Then we find the maximal value of S, $S[j_{max}, k_{max}]$. If it is greater than our threshold (0.5 in our case), We consider that j_{max} and k_{max} are the same and we repeat the process on the

matrix S with the lign j_{max} and the column k_{max} removed. If at the end, the matrix S is not empty, this means that every element of S is lower than the threshold. Therefore, these are new faces. We create a face for each remaining ligns in S.

3.1.2 • Improving tracking

Sometimes, a character will not be detected in a frame because their head is turning away or simply too far from the camera. This can cause huge performance issues. Indeed, if the face of a character is not detected on an isolated frame, then there will be two faces for this character: one before the missing frame and one after. This means that the face recognition will run twice instead of once. To prevent this, every so often, a function called *finish_process* runs.

If a face is not detected for less than 5 frames but was detected before and after, we consider that it is present in the scenes where it was not detected. Its position is computed through a simple linear interpolation. Obviously, this only works if there are no cuts in the scene.

On the opposite, sometimes irrelevant faces are detected. To resolve this, if a face is detected for less than 10 consecutive frames (including the additional interpolated frames described previously), then it is deleted.

3.1.3 • Tracking the mouth

To display bubbles, we need the position of the mouth. We used a simple approach to find the mouth position by averaging the position of landmarks associated to the mouth. Once again, this is coupled with the Taylor interpolation, used when the mouth position is not detected or computed for a given frame.

A major improvement in our algorithm was the assignment of bubbles to unknown speakers. In fact, sometimes the face recognition fails, which gives us two options: not assign the bubble to a mouth or assign the bubble to an unknown mouth. To improve the second option, we detect speaking using the movement of the lips. If the height of the mouth varies more than 15%, then the character is considered to be speaking. This is coupled with a filtering process similar to the tracking improvement: if a character stops speaking for less than 4 frames then we consider that they did not stop and if they speak for less than 8 frames, then we consider that they did not speak at all.

3.2 Finding the optimal placement of the bubble

One of the main challenges of this project was to find the optimal placement for a given subtitle. Let's first define a few rules:

- A bubble should not cover an important part of the frame (like a face) unless it is necessary

- The bubble should be close to the speaker but not so close that it touches its ear
- When possible, bubbles should be higher than the mouth
- A bubble must fit entirely on the screen and cannot hide another bubble.

The approach we chose for this task was to give a score to each pixel depending on how well it respects the conditions. We then compute the integrated image score and finally find the rectangle of a given size with the highest score thanks to the integrated image.

In order to better understand, let us review each contribution of the score separately.

3.2.1 • Boxes to avoid

For each object we want to avoid, we will assign a negative score to each pixel in its bounding box. We build the matrix score S_O, using a penalizer of 1.

In our version of the algorithm, the only objects to avoid are faces but ideally, we would use **YOLO** to detect several classes and assign a penalization to each class. For example, a face would have a strong penalization while a bike would have a small penalization. After trying YOLO, we did not use it as it was too slow and prevented our algorithm from running in real time.

3.2.2 • Distance to the mouth

We want the bubble to be close to the mouth but not too close. To enforce this, we compute the distance between each pixel and the mouth, normalize this distance between 0 and 1 and then use a scoring function inspired by the potential energy in terms of the radius of an orbiting object. After fine tuning, this is the function:

$$S_D(x) = 1.1681 * \left(\left(\frac{1}{0.5 * (0.18x + 0.2)} \right)^2 - 0.1. \frac{1}{0.5 * (0.18x + 0.2)} * (1 - x) \right)$$

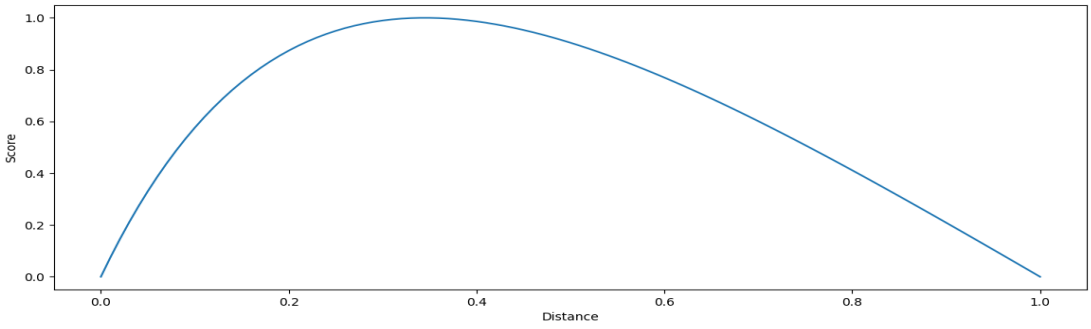


Figure 2: Score in terms of the normalized distance to the mouth

3.2.3 • Preferred position

To have our frames look even more like comics, we also want to give better scores to positions higher than the mouth and on the side of the head. Additionally, we want to avoid the borders and the direct top of the head. We will denote x_{head} and y_{head} as the starting position of the head bounding box. Additionally, we will add an horizontal margin ϵ, and κ the value at $y = y_{head}$.

$$f_x(x) = \begin{cases} 0.5 * \left(1 + cos\left(2\pi * \frac{x - x_{head} - \epsilon.x_{width}}{(1+2.\epsilon)x_{width}}\right)\right) & \text{if } x \in [x_{head} - \epsilon, x_{head} + 2\epsilon + x_{width}] \\ 1 & \text{otherwise} \end{cases}$$

$$f_y(y) = \begin{cases} sin\left((\pi - arcsin(\kappa)) * \frac{y}{y_{head}}\right) & \text{if } y \leq y_{head} \\ \kappa * \frac{1-y}{1-y_{head}} & \text{otherwise} \end{cases}$$

Which finally gives us the preferred score S_P: $S_P(x, y) = f_x(x) * f_y(y)$

3.2.4 • Combining everything

Finally, we compute the score S per pixel, combining the 3 previous scores, with the following formula : $S = S_P * S_D - \mu S_O$ with $\mu = 0.1$ *(reminder: S_O: boxes score, S_D: distance to the mouth score, S_P: preferred position score).*

3.2.5 • Example of result

Below, you will find the corresponding matrices **(where black means a high score and white a low score)**:

- S_O, the score associated to the boxes
- S_D, the score corresponding to the distance to the mouth
- S_P, the preferred position score
- S, the combined score
- The integrated score (sum over a rectangle of size $w = 0.2$ and $h = 0.25$)
- Final result - Bouding box of speaker - Mouth position - Obstacles - Position found for speech bubble

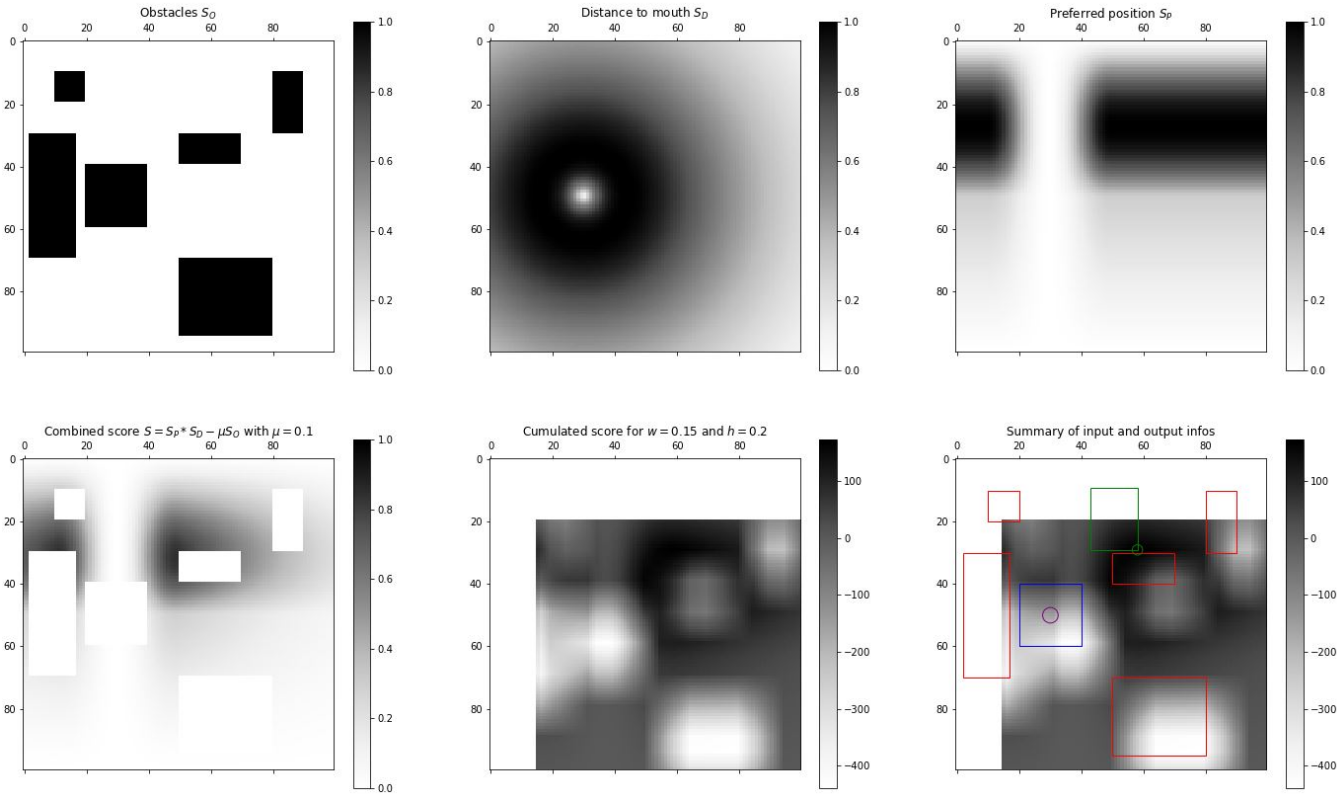


Figure 3: Score matrices for optimal position computing

3.3 Displaying the speech bubble

We create a Bubble class to implement our speech bubbles and draw the bubble (step (9) of the algorithm). Given a few parameters, it lets us display a bubble as a rounded rectangle, a triangle for the tail and text inside of it. To solve the problem of readability between frames, we display the rounded rectangle with the text at the same position for a given line. This means we only update the tail of the bubble in order to follow the mouth of the character between consecutive frames.

3.3.1 • Displaying the body and attach of the bubble

From every line of the subtitles, we gather the width of the bubble's body. We can estimate the height that will be needed to fit the text into that width. Using the placement of the bubble, we now have the center, the width and the height of the bubble's body, which we can display as a rounded rectangle.

We represent the tail of the bubble as a triangle. The tip of the triangle corresponds to the mouth of the character speaking. The position of the other vertices, which form the base of the triangle, is chosen to be close to the line that goes from the mouth to the center of the bubble.

Furthermore, the base of the triangle is stuck to one of the sides (left, right, up or down) of the bubble's body. We choose the side of the body depending on the position of the character's mouth relatively to the body's position, height and width.

When the face or mouth recognition fails to find the mouth of the right character (issue (10)), we choose to represent only the body of the bubble without a tail.

3.3.2 • Displaying the text inside the body of the bubble

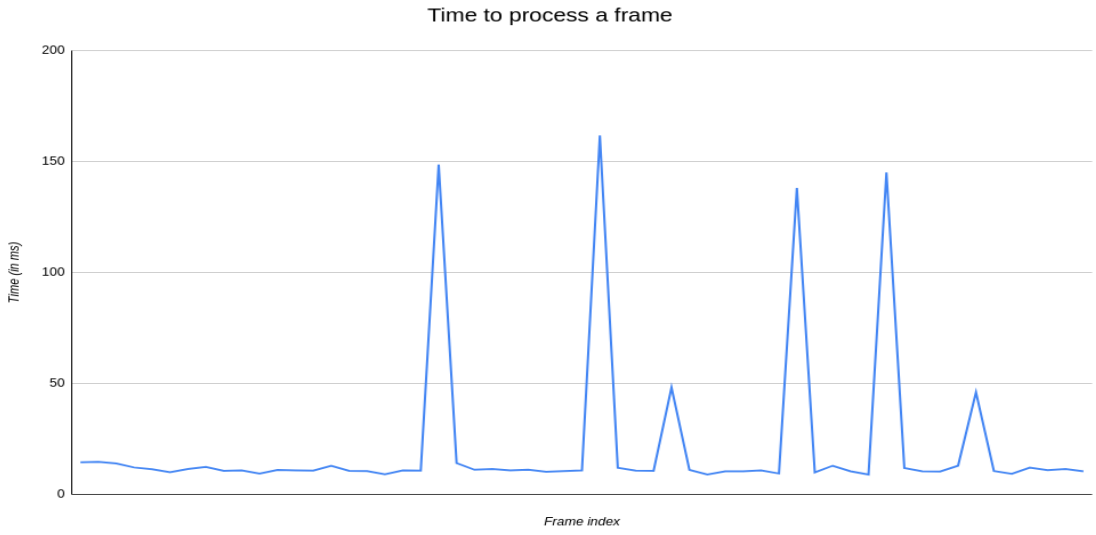
Once we know the width and height of the bubble's body, we need to display the text inside it. It should not go over the edge and it needs to be centered. We also need some processing of the text: typically, we look for the newline characters \n and \N and translate them into a new line for the text in our bubble.

We can now return the frame modified to include the speech bubble (11).

3.4 Optimizing the runtime

3.4.1 • Measures of the runtime

When running our script, we experienced big **freezes**. Let's have a look at the time needed to process a frame :



As we can see, there are many spikes. The big spikes correspond to the recognition of a character (which is really slow), and the small spikes correspond to the creation of a new bubble (finding the optimal position takes time).

On average, processing a frame takes 22.05ms on the previous test. Therefore, even when adding a few milliseconds to display the frame, we should be able to display the TV show at more than 30fps. However, with spikes of more than 150ms, the screen freezes.

3.4.2 • Parallel programming to the rescue

In order to solve our problem, we modified our code to make it runable in parallel. In fact, if we compute a certain number of frames in advance, then we can display and compute frames in parallel. That way, since the average time required to compute a frame is lower than 1fps, we will be able to compute the frames fast enough. Thus, when computing and displaying are run in parallel, the face recognition process won't block the display anymore.

To apply this parallel programming approach, we build 2 **queues**, a *pending* queue with all the frames computed, ready to be displayed, and a *processing* queue with frames being processed in parallel. Before displaying any frame, we wait for at least 150 frames to be pending, and we do not store more than 250 frames so as to not overload the memory.

In reality, to optimize memory consumption, the frame processing does not exactly return frames as it would be expensive in memory to store 250 high resolution frames in the RAM. Therefore, only key information are stored, such as faces objects with the key positions, name and associated speech.

With this improvement, we added a display for a few pieces of information such as the frame rate and the number of frames pending, as visible in the example below. We were able to maintain 4 frames per seconds.

Figure 4: Video player with information display

4 RESULTS

4.1 Qualitative Analysis

Our algorithm works pretty well. Most of the time, it succeeds in recognizing the mouth of the speaker and displays the bubble in a place that makes sense in the picture. For each frame, the mouth of the speaker is in red while the mouth of other characters or unknown people is in blue (see example below on the left). When the mouth of the speaker is not found, no bubble tail is displayed (see example below on the right).

Figure 5: Example of two frames with their subtitles as speech bubbles

4.2 Statisctics on a one minute video

As a benchmark of our algorithm, we will run it on a one-minute extract of Brooklyn 99, episode 8 of season 4, *the Monty-Hall issue*.

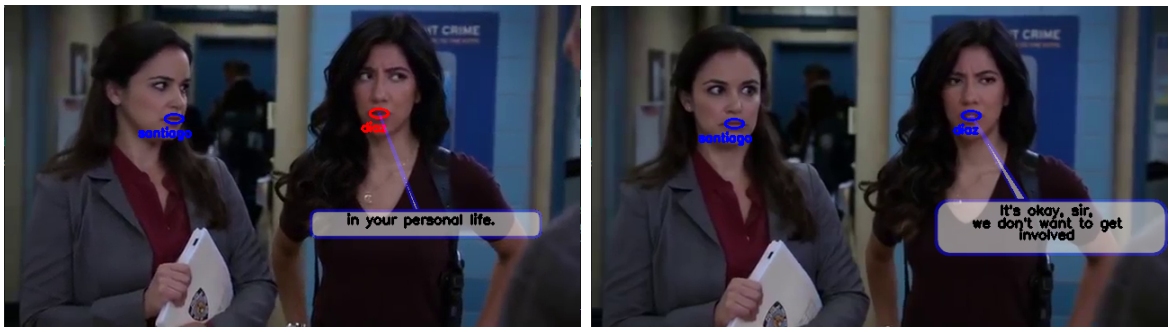


Figure 6: Extract of the Monty-Hall issue with working detection and recognition

Out of 17 lines (and thus 17 speech bubbles):

- 12 are associated to the right mouth, including 4 that recognize the character and display the right name. The algorithm fails in recognizing the character that is talking because of the mouth tracking: after a cut, it tends to associate a new mouth to a character whose mouth was more or less in the same position
- 2 are associated to no mouth at all, which is the expected behavious since the character speaking is not really visible.
- 3 are first associated to the wrong mouth before switching over to the right one. This happens when characters are out of range when they start speaking and the algorithm gives their lines to a character who just stopped talking. Then, after a cut, they appear and are given the line.

In terms of runtime, we were able to run this video at around 24fps, the original framerate. On slower computers of videos requiring a higher framerate, there are solutions :

- Reduce the resolution of the frame used for computation
- Compute only one frame over two. In fact, thanks to the interpolation used, the code can directly be used even if frames are skipped.
- Optimize the code even more. In fact, the recognition runs on characters that are not speaking, which is good to display our results but takes time.

4.3 Limits

The face recognition algorithm sometimes assigns the wrong character name to a character on the screen. For example, below, Raymond Holt is recognized as Amy Santiago. to solve this issue, we have defined confusions. As an example Kevin and Diaz are often mismatched, therefore if one of the two is in the frame, let's say Diaz, and Kevin is speaking but was not recognized then we assign the bubble to Diaz. This of course requires manual work, which we want to avoid and can lead to issues if the speaker is not in the scene.

Figure 7: Limiting case: wrong character detected

We did not take video cuts into account in our algorithm. This first means that mouth tracking continues after a video cut (when it should be reset), which caused 8/12 speaking characters to have the wrong name in our statistical analysis. Secondly, it means that when a character appears after a cut but their lines start before the cut, the algorithm tends to attribute the lines to a character that was present before the cut. Case in point below:

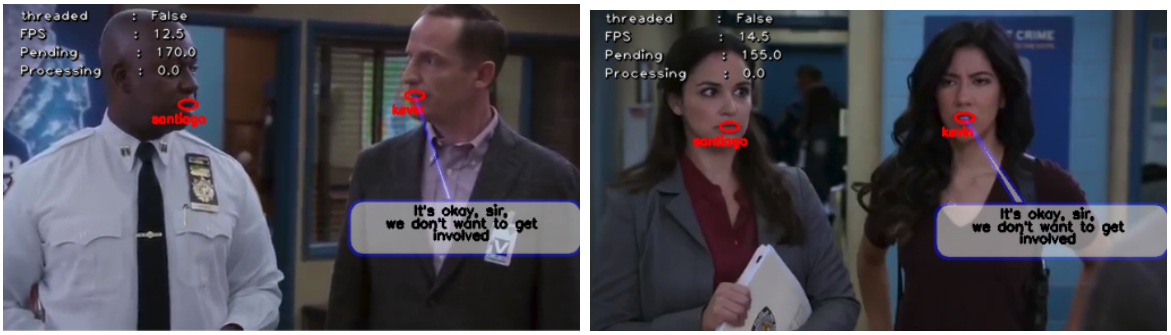


Figure 8: Limiting case around cuts: Kevin appearing to say Rosa's line before the cut

When several characters are speaking at the same time, our algorithm still works: bubbles are just added one after the other so that they don't touch. However, this means that the first bubble is not placed depending on other bubbles; therefore it will find its optimal placement not considering other bubbles, causing subsequent bubbles to also be placed in unsatisfactory positions. An example of this can be seen below, where the bubble tails intersect when it would have been preferable for each bubble to be under the character that says it.

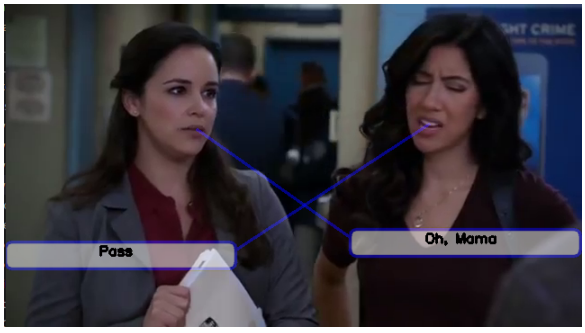


Figure 9: Limiting case: sub-optimal bubble placement

Our methods need to recognize characters and therefore to have a baseline. In the example of Brooklyn 99, only 5 pictures per characters were provided, but the fact remains we need to provide pictures for each character talking in the scene. In a future where our algorithm gets democratized, we could imagine that an image file for the TV show would be downloadable just like subtitles files. We could even imagine automatic fetching based on the actor's name and a tool like Amazon X-Ray. Without this, our algorithm cannot recognize characters, like in the picture below, and therefore assigns bubbles to random speakers.

Figure 10: Limiting case: unknown characters

Finally, our algorithm works as long as the detection works well. This means that if characters are far away in the scene, they won't be detected. Additionally, if the scene changes quickly like in an action movie, then not only will the detection struggle but the frequent cuts will cause the recognition to trigger often, causing frame drops.

Therefore our algorithm works well when the speaker is visible (unlike voice-over), and the action limited. It is well suited for dialogues between characters.