Overview

The final version of this script will find all MP4 and MTS files in a given folder, create audio files from the first 15s (can be adjusted), transcribe the audio files, and attempt to rename the movie files based on the transcript.

The current version creates audio files, but instead of renaming the movie files, prints final filenames and issues encountered to one of 3 different log files.

Installation

The *installer.ps1* script in the folder called **python** will download and install python and all required libraries.

Copy the **python** folder to your desired location (for example, your main system drive - usually `C:\`). Run the script by right-clicking on *installer.ps1* and choosing *Run with PowerShell* from the menu.

Description of files in the Transcriber folder

The **Transcriber** sub-folder within the python folder contains the script, helper functions, and files required to successfully run the script. Here is a list of the files that you should find there after running the installer script from the previous step. Other than any included movie files, they are all text files and can be edited using Notepad or Wordpad. However, they will be more legible and easier to work with if you use a dedicated text editor like Notepad++, Emacs, or Vim.

- transcriber.py - this script puts all the steps leading up to renaming the files together. A brief summary of the main steps involved will be provided later.
- helpers.py - this contains a list of functions that are repeatedly called by the main script. In general, these are mostly implementation details and will not require any editing.
- dogs.json - a list of the dogs that the script might encounter in the transcripts. Dogs can have multiple entries if the speech recognition service transcribes their names in different ways. For example, if the name **Ugo** is transcribed as *You go* and *Ugo*, 2 entries will be required -

> "You go": "Ugo"
>
> and
>
> "Ugo": "Ugo"
>
> The entry on the left specifies what the script will encounter in the transcript and the one on the right tells the script what label it should use for naming the file.

- locations.json - Similar to dogs.json, but for possible locations. Note that for dogs, exact matches are required; for locations and activities, approximate matches will suffice. This will be explained in more detail in the section on fuzzy matching.
- activities.json - list of activities that the script should use. If an activity is not in this list, the corresponding movie file will not be renamed. This can be fixed by simply adding the activity (or location or dog) to the activities.json (or locations.json or dogs.json) file.
- shortcut.cmd - a batch file that you can copy/move to your desktop. Double clicking this should execute the script once it is configured properly (see the section on Creating a Shortcut).
- Example movie files - These are included just to ensure that the script is working. These are not required and can be safely removed.
- README.md and README.pdf files - The md file is a markdown file which can be converted to a pdf using online tools. Feel free to edit these files in any way you think will be useful. To make edits, open up the md file in a text editor, make any changes and then use online "markdown to pdf converters" to retrieve the pdf.

Configuration

The only thing that needs to be setup before you run the script is the folder that it will search for movie files. By default, it is set to the folder that the script is in. To change it, open the transcriber.py file with your editor of choice, find the line that says `folder = '.'` and change the `.` to the directory that your videos are in. For example, if your videos are in `V:\Videos`, this line should read

```
folder = 'V:\Videos'
```

Running the script

To run the script from PowerShell, click on the Start menu and type "powershell". The application should come up in the search results. Once you have opened it, navigate to the folder that the script is in using the `cd` command. For example, if the script is in `C:\python\Transcriber`, you would type

```
cd C:\python\Transcriber
```

and then press 'Enter'.

PowerShell can auto-complete commands, so you can type part of a folder or command name and then press the "Tab" key to have PowerShell auto-complete what you were typing.

Next, assuming that the installation has been successful, you should be able to run the script by typing

```
..\python.exe .\transcriber.py
```

The '..' tells PowerShell to look in the previous folder and the '.' is for the current folder.

If you want to avoid this and create a shortcut to run the script, read on.

Creating a shortcut

Included in the Transcriber folder is a file called **shortcut.cmd**. Copy or move this file to your Desktop or wherever you want the shortcut to go. By default, it will look for the script in `C:\python\Transcriber`. To change this, open the file with a text editor and change the first line. For example, if you have installed the script to `V:\Videos`, the first line should read

```
cd V:\Videos\python\Transcriber
```

Save your changes.

Understanding the output

Once the script finishes running, you will find a few new files in your videos directory. The '.log' files are the important ones and will tell you what the script did. Here is a description of the various output files -

- Audio files - Each movie file will have a corresponding audio file (.wav extension) of length 15s. The final version will delete these files, but I am having the testing version keep them in case we need them for debugging.
- matched.log - This log file will provide you with a list of all the movie files where the name determined by the script was the same as the file's original name. As far as testing goes, these are the files that the script is working well on and they should need no further attention.
- unmatched.log - From a debugging perspective, this is the most important file. There will be 2 lines for each file. The first line will tell you the original name and the new name determined by the script. The second line will provide you with the transcript of the audio file determined by google's speech recognition service.
- unsuccessful.log - These will be the files that the script was not able to determine a name for. If the speech recognition service was unable to transcribe anything, the log for the file will only take up 1 line. The line will give you the file name and say that nothing could be transcribed. If the name determination failed for any other reason, the log will consist of 3 lines - the first line will provide the name of the file, the second will provide the transcript, and the third will tell you why no name could be determined. The only possible reasons are that either the dog's name, the location name, or the activity could not be determined using their corresponding json files. Testing-wise, I think this is less important and we should first try and get the unmatched files to either the matched or unsuccessful state.

Brief summary of the script

1. Find all mp4 and mts files in the specified directory.
2. Create wav files from the first 15 seconds of the movie files.
3. Load the dogs.json, activites.json, and locations.json files to check the transcripts against.
4. Send data from the audio files to google's speech recognition service and receive the transcript (if speed is a concern, we can look into using offline speech recognition services as well).
5. Find the appropriate labels using the json files.
    1. Dog names have te be exact matches because many of the dogs' names are short. Using approximate matching would not work. For example, if we used approximate matching, ***Sky*** would be found in *Kentuc**ky** fried chicken*.
    2. Since location and activity names are longer, I am using approximate matching for them. For example, if the transcript contains the word "Margaret's", but the locations json file only has "Margrit's" listed, "Margrit's" should still be a match. (This specific example actually wouldn't work because the threshold has to be set fairly high. Locations like "Buses" and "Annex" are short, so the threshold for matching has to be set high). For more details on this, refer to the section title **Fuzzy Matching**.
6. Use the movie file's modification date to figure out the date the video was created on. I decided to use this approach because it was simpler. Since different people say the date differently, parsing the transcript for all possible date formats would be difficult, but probably not impossible. I also figured that people might occasionally get the date wrong. The only way the modification date would provide an incorrect label would be if the movie file was edited in any way before re-naming it.
7. If a name could be determined, write it to either the matched or unmatched file depending on whether the determined name matches the original name.
8. If no name was determined, write relevant details to the unsuccessful file.

Suggested Testing Strategy

Create a new folder specifically for testing. Re-direct the script to search this folder (see the section on **Configuration**). Move 5-10 previously named files at a time and run the script on them. If the files in the 'unmatched.log' file are incorrectly named (beyond simple formatting issues), save the transcripts for those files so we can figure out how to fix it. If names don't match because of different formats, those are easy fixes and you would just need to tell me what the format needs to be.

Known bugs

1. The installer script requires a TLS version greater than the one PowerShell uses (used to use?) by default. Need to write in a check for this so that the script doesn't terminate if the incorrect version is being used.
2. Since there is overlap in the locations and activities (eg. "Agility Yard" in locations, "Agility" and "Agility Foundations" in activities), the script will sometimes come up with incorrect names. Suppose the transcript is "Juel Agility Yard Obedience July 23rd 2020", the script would pick up both "Agility" and "Obedience" as activities. The filename would then be "Juel, Agility and Obedience, Agility Yard, 7-23-2020.MP4". Working on a fix for this.

Fuzzy Matching

To implement approximate matching (fuzzy matching) for locations and activities, I am using algorithms that employ the Leveshtein distance as a measure of similarity between two strings. Similarity is computed by looking at the cost incurred when replacing one string with another. Insertions, deletions, and substitutions have associated costs. The costs are: * Insertion - 1 * Deletion - 1 * Substitution - 2

Once the total cost of replacing one string with another is calculated, the distance is calculated as:

```
distance = 100 - cost * 100 / (length(string 1) + length(string 2))
```

A score of 100 would mean that the two strings are exactly the same, while a score of 0 would mean that every letter in one word is different from the letters in the other one.

For example, suppose we are comparing "buses" with "uses". There is one deletion - the letter 'b'. The length of buses is 5 and that of uses is 4. So, the distance is

```
100 - 1 * 100 / 9
= 100 - 100 / 9
= 89
```

approximately.

The specific algorithm I am using compares a string to another of the same size. If the sizes are different, it shortens the longer string to the length of the shorter one. Let's use the example I used in the "Brief summary ..." section. We were comparing "Margrit's" with "Margaret's". The algorithm first strips all non-numbers and non-letters (we can override this if necessary), so it would compare "Margrits" with "Margarets". Since "Margarets" is longer, it would try comparing "Margrits" to "Margaret" and to "argarets".

First, let's deal with "Margrits" vs. "Margaret". * Deletions - 1 ('a') * Substitutions - 1 ('e -> i') * Insertions - 1 which gives us a total cost of 4 (2 for the substitution).

```
distance = 100 - 400/16
         = 75
```

For "Margrits" vs. "argarets" * Deletions - 1 ('a') * Substitutions - 1 ('e -> i') * Insertions - 1 ('M')

The cost and the distance is the same as in the earlier case. If the costs for the two cases happen to be different, the algorithm chooses the better match and reports the score for that match.

The reason why this approach cannot be used for dogs' names is because many of them are short. So, comparing ***Sky*** to *Kentuc**ky** fried chicken* would result in comparing "Sky" to "cky" (this comparison produces the highest score). There is one substitution, so the score would end up being 67 (approximately). To use this approach then, we would have to set the threshold quite low and would end up detecting spurious matches. For locations and activities, the length of the names should prevents this from happening - we can set the threshold quite high (currently at 91) and that should still allow us to pick up slight mis-matches.