Simple Audio Indexer Documentation

Release 1.0.0

Alireza Rafiei

Dec 28, 2020

Contents:

Simple Audio Indexer

Simple Audio Indexer, (or sai, to be shorter!) is a Python library and command-line tool that enables one to search for a word or a phrase within an audio file.

Based on that simple core, it generalizes the searching process and lets one to search for multiple queries within multiple audio files of any size.

It also provides advanced searching abilities by allowing the developer to either use the currently implemented control structures on the search methods, or for the developer to define her own patterns in regex and match them to contents of the audio file.

sai is open-source, licensed under Apache v2.0.

CHAPTER 1

Installation

There are two main ways to install sai:

1. If you're on a Unix(-like) system, say Linux or OS X, then you can do a full/native installation.
2. If you're on a Windows system, or for some reason don't want to install natively, you may use the software within a Docker image.

1.1 Native Installation

You're going to need to get IBM Watson credentials, install sox and finally install sai.

1.1.1 First step: IBM Credentials

You need a valid username and password for IBM Watson's Speech to Text API. You may sign up here. After you've created your accout, make an app that uses Speech to text service. Go the settings and save your credentials.

The process has been explained in detail in here

1.1.2 Second Step: Installing sox

You need to install sox on your system. We'll use sox to process the audio.

If you're on a *Linux* system, it should probably be in your distro's repository. If you're using Ubuntu (or similar), you may install by entering the command below in a terminal:

```
sudo apt-get install sox
```

If you're on *OS X*, then choose the most recent version from the sox 's official repo and install it on your system. The link is here.

If you're using homebrew, however, you could just enter the command below in a terminal:

```
brew install sox
```

1.1.3 Third Step: Installing SAI

You should be installing this library via Python's pip. Enter the command below in a terminal:

```
pip install SimpleAudioIndexer
```

Note that if you wish to run the unit tests, you need to install pytest (and preferably tox as well).

That's it! If everything was okay, you should be having it on your system. To verify, enter in your terminal:

```
sai -h
```

You may also enter in a Python shell:

```
>>> from SimpleAudioIndexer import SimpleAudioIndexer as sai
```

If you didn't see any error messages, then sai is successfully installed!

That's it! you've installed sai successfully!

1.2 Offline indexing with CMU Pocketsphinx

You have an option to use CMU Pocketsphinx as your audio indexer. Note that the quality of Pocketsphinx is at "pre-alpha" level which means almsot never you'd see a result that's perfectly accurate.

Only use this option if you don't want your files being uploaded to Watson's servers, or you're on Windows and don't want to go in the *Docker route*.

1.2.1 First step: Installing ffmpeg

You need to install ffmpeg to regularize the encoding of your audio files.

If you're on Linux, it should probably be in your repositories. You may download ffmpeg on Ubuntu via

```
sudo apt-get install ffmpeg
```

If you're on Mac, you may either go to ffmpeg 's website and download it, or install it via homebrew by entering:

```
brew install ffmpeg
```

1.2.2 Second step: Installing Pocketsphinx

Use the official guide here to compile it. The guide is relatively straightforward.

Note that unless you know aboslutely what you're doing, don't install prepackaged versions e.g. from your distributions repositories etc.

1.2.3 Third step: Installing everything else

Install sox and sai natively, as it was described previously!

1.3 Docker route

If you're on a Windows system, or for some reason don't want to install natively you may run sai within a docker container.

We don't recommend that you choose the docker route if you have a choice to do a native install. Docker containers are intended to run a single process and will stop as soon as their job is complete.

Our image, however, will run a process that never ends which in turn would enable you to get a terminal in that container.

We assume that you have docker installed and functional on your system.

Download the Dockerfile.txt from the sai 's repository.

Open up a terminal and *cd* into the directory that contains the docker file. Then, enter the command below:

```
docker build -t sai-docker .
```

Note that by running building our docker image, you'd be downloading a lot of intermediary stuff including Ubuntu and a new build of Python. That means, you should have at least 500MB available.

Assuming the build was successfull, then enter the command below to run it:

```
docker run sai-docker
```

Now open up a new terminal and enter the command below:

```
docker ps -a
```

Now copy the Container-ID of *sai-docker*. Then, in that new terminal enter:

```
docker exec -i -t CONTAINER-ID /bin/bash
```

Right now you should be having shell access within *sai-docker* container and should be able to run *sai* in the command line or import it in a python REPL.

To stop the docker process, exit the shell you've got in the container and open up a new terminal in your system and enter:

```
docker rm -rf CONTAINER-ID
```

1.4 Uninstall

If for any reason you wish to install sai, fear not! It's quite simple.

1.4.1 Uninstall natively

If you've installed sai natively on your system, then you may just open up a command line and enter:

```
pip uninstall SimpleAudioIndexer
```

Depending on your operating system, uninstallation method of sox would be different. If you're on Ubuntu, you may just enter:

```
sudo apt-get remove sox && sudo apt-get autoremove
```

If you were on OS X and used homebrew, you may open up a terminal and enter:

```
brew uninstall sox
```

If however you've installed sox via their repo, then it'd be just a simple drag and drop wherever you've installed it!

That's it! You've uninstalled sai successfully!

1.4.2 Uninstalling CMU Pocketsphinx

You may uninstall sox and sai like it was described above. For uninstalling ffmpeg, proceed similarly to sox i.e. if you're on an Ubuntu

:: sudo apt uninstall ffmpeg

or on Mac using homebrew

```
brew uninstall ffmpeg
```

To uninstall CMU Sphinx, go into the directory which you've compiled it and enter:

```
make uninstall
```

And then remove that directory.

1.4.3 Uninstall the Docker version

If you've installed sai from the dockerfile.txt found at the repo, then you may just open up a terminal and enter:

```
docker rmi sai-docker
```

Note an Ubuntu image would be installed alongside sai-docker as well. You may remove that similarly.

CHAPTER 2

Usage

There are basically two ways to use sai:

1. *As a command line script*.
2. *As a library for developers*. (recommended)

2.1 As a command line script

Note that currently the command-line script is very limited in its functionality and not all the available methods have been implemented for the command-line interface.

We assume you have sai installed and have IBM Watson's credentials ready (if not, you may read the installation guide here).

2.1.1 The help command

Open up a terminal. Enter:

```
sai -h
```

The result would be a list of all the implemented commands for the command-line script.

2.1.2 Timestamps

You can either choose to use Watson as your speech to text engine or Pocketsphinx.

Say your choice is Watson, then enter the command below (replace *USERNAME* and *PASSWORD* with your Watson credentials) and replace *SRC_DIR* with the absolute path to the directory in which your audio files are located):

```
sai --mode "ibm" --username_ibm USERNAME --password_ibm PASSWORD --src_dir SRC_DIR --
↪timestamps
```

The command above should give you the timestamps of all the words within the audio.

Note that the switches *–mode* and *–src_dir* are required for anything you want to do. If your choice of mode is *ibm*, then the switches *username_ibm* and *password_ibm* would be required as well.

There are some optional switches as well. You've used *–timestamps* to get timestamps, however you may use *–search* if you want to search for something, or *–regexp* to search with regular expressions etc.

For the rest of this tutorial we assume that your choice of mode is *ibm*. If your choice is *cmu*, then you no longer need to ibm switches, but everything else remains the same.

2.1.3 Search Commands

Say you want to search for the string "apple", then your command would be:

```
sai --mode ibm --src_dir SRC_DIR --username_ibm USERNAME --password_ibm PASSWORD --
↪search "apple"
```

You may also search for a regex pattern via the switch *–regexp*:

```
sai --mode ibm --src_dir SRC_DIR --username_ibm USERNAME --password_ibm PASSWORD --
↪regexp " [a-z][a-z] "
```

Note that you cannot simultaneously use *-s* and *-r*!

The default language is American English. If you want to use another language/accent, then use the switch *–language*. For the list of all models, see the reference here.

Note that *–language* switch doesn't work for *cmu* mode.

2.1.4 Saving & Loading indexed data

The last thing worth mentioning, is that, say your audio files are big enough that you don't want to spend time indexing them every time you run the script, then you should use the switch *–save-data* followed by an absolute path to a file into which the indexed data would be written. If such a file doesn't exist, it'll be created.

For example, say I want to write the indexed data into my Documents directory with the name *indexed_data*. Then the command would be:

```
sai --mode ibm --src_dir SRC_DIR --username_ibm USERNAME --password_ibm PASSWORD --
↪save-data ABSOLUTE_PATH_TO_A_FILE_TO_BE_CREATED/indexed_data
```

Next time that I want to search the those audio files, I'll enter:

```
sai --mode ibm --load_data ABOSLUTE_PATH_INDEXED_DATA --search "stuff"
```

Note that whenever you're loading your data, you no longer have to enter your username and password and src_dir, but you still need to specify your mode.

2.2 As a Python library

We assume you have sai installed and have IBM Watson's credentials ready (if not, you may read the installation guide here).

2.2.1 Basics

You should put your audio files inside a single directory. Where ever we say *SRC_DIR*, you should replace it via the absolute path of that directory. You should also replace *USERNAME* and *PASSWORD* with your Watson credentials.

Note that the format of your audio files must be *wav* (Specific encodings wont matter). However if your file is not *wav*, it'll be ignored.

Also note that, your audio files must end in *.wav*. While some operating system allow you to store files without explicit format decleration, it won't always be reliable to look at the header of the audio files or guess the format.

There's one class that you need to import:

```
>>> from SimpleAudioIndexer import SimpleAudioIndexer as sai
```

Afterwards, you should create an instance of *sai*

```
>>> indexer = sai(mode="ibm", src_dir="SRC_DIR", username_ibm="USERNAME", password_
↪ibm="PASSWORD")
```

Note that if you choose your mode to be *cmu*, then you no longer have to provide username and passwords of your Watson account.

2.2.2 Indexing

Now you may index all the available audio files by calling *index_audio* method:

```
>>> indexer.index_audio()
```

This method automatically created two directories *filtered* and *staging* within your *src_dir* to handle intermediarry files.

Also, you could also just index a particular audio file. Say you only wish to index *SRC_DIR/target.wav*, then:

```
>>> indexer.index_audio(basename=target.wav)
```

For more information on all arguments of this method (including other languages or accuracy etc.) read the API reference here

2.2.3 Saving & Loading Indexed data

index_audio method, transfers *wav* files into the *filtered* directory. Then, checks the size of the audio file and splits it if they are sufficiently large and moves them to *staging* directory and finally reads and sends a request to Watson.

Say you've done all of that and the next time you don't want to make that request. Then save your data:

```
>>> indexer.save_indexed_audio("{}/indexed_audio".format(indexer.src_dir))
```

Afterwards, all the timestamps of the audios would be saved in *SRC_DIR/indexed_audio.txt*. Next time, instead of calling *index_audio* method, do:

```
>>> indexer.load_indexed_audio("{}/indexed_audio.txt".format(indexer.src_dir))
```

2.2.4 Timestamps and time regularizations

After you've indexed audio, the timestamps for each word would be saved within a private attribute. They should not be accessed since if the audio files were large and they were splitted, then the timing won't be correct.

The time corrected/regularized however can be accessed via *get_timestamps* method. Say there are two audio files in *SRC_DIR* called *audio.wav* and *another.wav*. Then the timestamps would look something like below:

```
>>> print(indexer.get_timestamps())
{"audio.wav": [["hello", 0.01, 0.05], ["how", 0.05, 0.08], ["are", 0.08, 0.11],
["you", 0.11, 0.14]], "another": [["yo", 0.01, 0.02]]}
```

Basically, the output is a dictionary whose keys are the audio files and the outputs are a list of word blocks. A word block is a list whose first element is a word, second element is the starting second and the third (and last) element is the ending second of that word.

2.2.5 Searching methods

Now, search methods all use the *get_timestamps* internally. Say after indexing, you finally wanted to do a search.

You could have a searching generator:

```
>>> searcher = indexer.search_gen(query="hello")
# If you're on python 2.7, instead of below, do print searcher.next()
>>> print(next(searcher))
{"Query": "hello", "File Name": "audio.wav", "Result": [(0.01, 0.05)]
```

So in the example above, *SRC_DIR/audio.wav* is at least 0.14 seconds long and contains 4 words: "hello", "how", "are", "you".

Now there are quite a few more arguments implemented for search_gen. Say you wanted your search to be case sensitive (by default it's not). Or, say you wanted to look for a phrase but there's a timing gap and the indexer didn't pick it up right, you could specify *timing_error*. Or, say some word is completely missed, then you could specify *missing_word_tolerance* etc.

For a full list, see the API reference here

You could also call *search_all* method to have search for a list of queries within all the audio files:

```
>>> print(indexer.search_all(queries=["hello", "yo"]))
{"hello": {"audio.wav": [(0.01, 0.05)]}, {"yo": {"another.wav": [(0.01, 0.02)]}}}
```

The same arguments that were applicable for *search_gen* are applicable for *search_all*.

Finally, you could do a regex search!

```
>>> print(indexer.search_regexp(pattern=" [a-z][a-z][a-z] ")
{"are": {"audio.wav": [(0.08, 0.11)]}, "how": {"audio.wav": [(0.05, 0.08)]},
"you": {"audio.wav": [(0.11, 0.14)]}}
```

Note that anything that can be done via the implemented word-based control structures over *search_gen* can be done via regex pattern matching (albeit maybe nontrivial to write the correct pattern).

The open ended nature of *search_regexp* is intended to compliment *search_gen*.

That's it! You know enough to get started. I recemmend taking a look at API reference here to learn more about other methods that have been implemented.

CHAPTER 3

API Reference

class SimpleAudioIndexer.**SimpleAudioIndexer**(*src_dir*, *mode*, *username_ibm=None*, *password_ibm=None*, *ibm_api_limit_bytes=100000000*, *verbose=False*, *needed_directories=set(['filtered', 'staging'])*)

Indexes audio and searches for a string within it or matches a regex pattern.

Audio files that are intended to be indexed should be in wav format, placed in a same directory and the absolute path to that directory should be passed as *src_dir* upon initialization.

Call the method *index_audio* (which results in calling *index_audio_ibm* or *index_audio_cmu* based on the given mode) prior to searching or accessing timestamps, unless you have saved the data for your previously indexed audio (in that case, *load_indexed_audio* method must be used)

You may see timestamps of the words that have been indexed so far sorted by audio files and the time of their occurance, by calling the method *get_audio_timestamps*.

You may saved the indexed audio data (which is basically just the time- regularized timestamps) via *save_indexed_audio* method and load it back via *load_indexed_audio*

Do exhustive search with the method *search_all*, do iterative search with the method *search_gen* or do regex based search with the method *search_regexp*

For more information see the docs and read usage guide.

Attributes

mode [{"ibm", "cmu"}] specifying whether speech to text engine is IBM's Watson or Pocketsphinx.

src_dir [str] Absolute path to the source directory of audio files such that the absolute path of the audio that'll be indexed would be *src_dir/audio_file.wav*

verbose [bool, optional] *True* if progress needs to be printed. Default is *False*.

ibm_api_limit_bytes [int, optional] It holds the API limitation of Watson speech api http sessionless which is 100Mbs. Default is 100000000.

Methods

get_mode()	
get_username_ibm()	
set_username_ibm()	
get_password_ibm()	
set_password_ibm()	
get_verbosity()	
set_verbosity()	
get_timestamps()	Returns a corrected dictionary whose key is the original file name and whose value is a list of words and their beginning and ending time. It accounts for large files and does the timing calculations to return the correct result.
get_errors()	Returns a dictionary that has all the erros that have occured while processing the audio file. Dictionary contains time of error, file that had the error and the actual error.
_index_audio_ibm(name=None, continuous=True, model="en-US_BroadbandModel",	word_confidence=True, word_alternatives_threshold=0.9, profanity_filter_for_US_results=False) Implements a searching-suitable interface for the Watson API
_index_audio_cmu(name=None)	Implements an experimental interface for the CMu Pocketsphinx
index_audio(*args, **kwargs)	Returns a corrected dictionary whose key is the original file name and whose value is a list of words and their beginning and ending time. It accounts for large files and does the timing calculations to return the correct result.
save_indexed_audio(indexed_audio_file_abs_path)	
load_indexed_audio(indexed_audio_file_abs_path)	
search_gen(query, audio_basename=None, case_sensitive=False,	subsequence=False, supersequence=False, timing_error=0.0, anagram=False, missing_word_tolerance=0) A generator which returns a valid search result at each iteraiton.
search_all(queries, audio_basename=None, case_sensitive=False,	subsequence=False, supersequence=False, timing_error=0.0, anagram=False, missing_word_tolerance=0) Returns a dictionary of all results of all of the queries for either all of the audio files or the *audio_basename*.
search_regexp(pattern, audio_basename=None)	Returns a dictionary of all results which matched *pattern* for either all of the audio files or the *auio_basename*

get_mode(*self*)

Returns whether the instance is initialized with *ibm* or *cmu* mode.

Returns

str

get_username_ibm(*self*)

Returns

str, None Returns *str* if mode is *ibm*, else *None*

set_username_ibm(*self*, *username_ibm*)

Parameters

username_ibm [str]

Raises

Exception If mode is not *ibm*

get_password_ibm(*self*)

Returns

str, None Returns *str* if mode is *ibm*, else *None*

set_password_ibm(*self*, *password_ibm*)

Parameters

password_ibm [str]

Raises

Exception If mode is not *ibm*

get_verbosity(*self*)

Returns whether the instance is initialized to be quite or loud while processing audio files.

Returns

bool True for being verbose.

set_verbosity(*self*, *pred*)

Parameters

pred [bool]

get_timestamps(*self*)

Returns a dictionary whose keys are audio file basenames and whose values are a list of word blocks. In case the audio file was large enough to be splitted, it adds seconds to correct timing and in case the timestamp was manually loaded, it leaves it alone.

Returns

{str: [[str, float, float]]}

get_errors(*self*)

Returns a dictionary containing any errors while processing the audio files. Works for either mode.

Returns

{(float, str): any} The return is a dictionary whose keys are tuples whose first elements are the time of the error and whose second values are the audio file's name. The values of the dictionary are the actual errors.

index_audio(*self*, **args*, ***kwargs*)

Calls the correct indexer function based on the mode.

If mode is *ibm*, _indexer_audio_ibm is called which is an interface for Watson. Note that some of the explaination of _indexer_audio_ibm's arguments is from [1]

If mode is *cmu*, _indexer_audio_cmu is called which is an interface for PocketSphinx Beware that the output would not be sufficiently accurate. Use this only if you don't want to upload your files to IBM.

Parameters

mode [{"ibm", "cmu"}]

basename [str, optional] A specific basename to be indexed and is placed in src_dir e.g *audio.wav*.

If *None* is selected, all the valid audio files would be indexed. Default is *None*.

replace_already_indexed [bool] *True*, To reindex some audio file that's already in the timestamps.

Default is *False*.

continuous [bool] Valid Only if mode is *ibm*

Indicates whether multiple final results that represent consecutive phrases separated by long pauses are returned. If true, such phrases are returned; if false (the default), recognition ends after the first end-of-speech (EOS) incident is detected.

Default is *True*.

model [{]

'ar-AR_BroadbandModel', 'en-UK_BroadbandModel' 'en-UK_NarrowbandModel', 'en-US_BroadbandModel', (the default) 'en-US_NarrowbandModel', 'es-ES_BroadbandModel', 'es-ES_NarrowbandModel', 'fr-FR_BroadbandModel', 'ja-JP_BroadbandModel', 'ja-JP_NarrowbandModel', 'pt-BR_BroadbandModel', 'pt-BR_NarrowbandModel', 'zh-CN_BroadbandModel', 'zh-CN_NarrowbandModel'

}

Valid Only if mode is *ibm*

The identifier of the model to be used for the recognition

Default is 'en-US_BroadbandModel'

word_confidence [bool] Valid Only if mode is *ibm*

Indicates whether a confidence measure in the range of 0 to 1 is returned for each word.

The default is True. (It's False in the original)

word_alternatives_threshold [numeric] Valid Only if mode is *ibm*

A confidence value that is the lower bound for identifying a hypothesis as a possible word alternative (also known as "Confusion Networks"). An alternative word is considered if its confidence is greater than or equal to the threshold. Specify a probability between 0 and 1 inclusive.

Default is *0.9*.

profanity_filter_for_US_results [bool] Valid Only if mode is *ibm*

Indicates whether profanity filtering is performed on the transcript. If true, the service filters profanity from all output by replacing inappropriate words with a series of asterisks.

If false, the service returns results with no censoring. Applies to US English transcription only.

Default is *False*.

Raises

OSError Valid only if mode is *cmu*.

If the output of pocketsphinx command results in an error.

References

Else if mode is *cmu*, then _index_audio_cmu would be called:

[1]

save_indexed_audio(*self*, *indexed_audio_file_abs_path*)
Writes the corrected timestamps to a file. Timestamps are a python dictionary.

Parameters

indexed_audio_file_abs_path [str]

load_indexed_audio(*self*, *indexed_audio_file_abs_path*)

Parameters

indexed_audio_file_abs_path [str]

search_gen(*self*, *query*, *audio_basename=None*, *case_sensitive=False*, *subsequence=False*, *supersequence=False*, *timing_error=0.0*, *anagram=False*, *missing_word_tolerance=0*)
A generator that searches for the *query* within the audiofiles of the src_dir.

Parameters

query [str] A string that'll be searched. It'll be splitted on spaces and then each word gets sequentially searched.

audio_basename [str, optional] Search only within the given audio_basename.

Default is *None*

case_sensitive [bool, optional] Default is *False*

subsequence [bool, optional] *True* if it's not needed for the exact word be detected and larger strings that contain the given one are fine.

If the query is a sentences with multiple words, it'll be considered for each word, not the whole sentence.

Default is *False*.

supersequence [bool, optional] *True* if it's not needed for the exact word be detected and smaller strings that are contained within the given one are fine.

If the query is a sentences with multiple words, it'll be considered for each word, not the whole sentence.

Default is *False*.

anagram [bool, optional] *True* if it's acceptable for a complete permutation of the word to be found. e.g. "abcde" would be acceptable for "edbac".

If the query is a sentences with multiple words, it'll be considered for each word, not the whole sentence.

Default is *False*.

timing_error [None or float, optional] Sometimes other words (almost always very small) would be detected between the words of the *query*. This parameter defines the timing difference/tolerance of the search.

Default is 0.0 i.e. No timing error is tolerated.

missing_word_tolerance [int, optional] The number of words that can be missed within the result. For example, if the query is "Some random text" and the tolerance value is *1*, then "Some text" would be a valid response. Note that the first and last words

cannot be missed. Also, there'll be an error if the value is more than the number of available words. For the example above, any value more than 1 would have given an error (since there's only one word i.e. "random" that can be missed)

Default is 0.

Yields

{"File Name": str, "Query": *query*, "Result": (float, float)} The result of the search is returned as a tuple which is the value of the "Result" key. The first element of the tuple is the starting second of *query* and the last element is the ending second of *query*

Raises

AssertionError If *missing_word_tolerance* value is more than the total number of words in the query minus 2 (since the first and the last word cannot be removed)

search_all(*self*, *queries*, *audio_basename=None*, *case_sensitive=False*, *subsequence=False*, *supersequence=False*, *timing_error=0.0*, *anagram=False*, *missing_word_tolerance=0*)

Returns a dictionary of all results of all of the queries for all of the audio files. All the specified parameters work per query.

Parameters

queries [[str] or str] A list of the strings that'll be searched. If type of queries is *str*, it'll be insterted into a list within the body of the method.

audio_basename [str, optional] Search only within the given audio_basename.

Default is *None*.

case_sensitive [bool] Default is *False*

subsequence [bool, optional] *True* if it's not needed for the exact word be detected and larger strings that contain the given one are fine.

If the query is a sentences with multiple words, it'll be considered for each word, not the whole sentence.

Default is *False*.

supersequence [bool, optional] *True* if it's not needed for the exact word be detected and smaller strings that are contained within the given one are fine.

If the query is a sentences with multiple words, it'll be considered for each word, not the whole sentence.

Default is *False*.

anagram [bool, optional] *True* if it's acceptable for a complete permutation of the word to be found. e.g. "abcde" would be acceptable for "edbac".

If the query is a sentences with multiple words, it'll be considered for each word, not the whole sentence.

Default is *False*.

timing_error [None or float, optional] Sometimes other words (almost always very small) would be detected between the words of the *query*. This parameter defines the timing difference/tolerance of the search.

Default is 0.0 i.e. No timing error is tolerated.

missing_word_tolerance [int, optional] The number of words that can be missed within the result. For example, if the query is "Some random text" and the tolerance value is *1*, then "Some text" would be a valid response. Note that the first and last words

cannot be missed. Also, there'll be an error if the value is more than the number of available words. For the example above, any value more than 1 would have given an error (since there's only one word i.e. "random" that can be missed)

Default is 0.

Returns

search_results [{str: {str: [(float, float)]}}] A dictionary whose keys are queries and whose values are dictionaries whose keys are all the audiofiles in which the query is present and whose values are a list whose elements are 2-tuples whose first element is the starting second of the query and whose values are the ending second. e.g. {"apple": {"fruits.wav" : [(1.1, 1.12)]}}

Raises

TypeError if *queries* is neither a list nor a str

search_regexp(*self*, *pattern*, *audio_basename=None*)

First joins the words of the word_blocks of timestamps with space, per audio_basename. Then matches *pattern* and calculates the index of the word_block where the first and last word of the matched result appears in. Then presents the output like *search_all* method.

Note that the leading and trailing spaces from the matched results would be removed while determining which word_block they belong to.

Parameters

pattern [str] A regex pattern.

audio_basename [str, optional] Search only within the given audio_basename.

Default is *False*.

Returns

search_results [{str: {str: [(float, float)]}}] A dictionary whose keys are queries and whose values are dictionaries whose keys are all the audiofiles in which the query is present and whose values are a list whose elements are 2-tuples whose first element is the starting second of the query and whose values are the ending second. e.g. {"apple": {"fruits.wav" : [(1.1, 1.12)]}}

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] : https://ibm.com/watson/developercloud/speech-to-text/api/v1/

Index

G

I

L

S