Hutts Verification Documentation

Release 0.1

Java the Hutts

Oct 18, 2017

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Hutts Verification is a Web API that performs electronic ID extraction and verification. The primary purpose of the system is to reduce the time that it takes to manually capture a user's personal information by automatically extracting it from the person's official identification documentation. This information includes both the facial and the textual data that is evident on such documentation. The system can then also use this extracted information to compare it to data that it is provided with in order to give a percentage match of the two sets of data.

Due to the main target market of Hutts Verification being developers, the system is designed as an easily pluggable API for any Python-based system. A server and a graphical interface is also provided so that a user can experiment with the system in a clean and simple manner without needing any previous development experience.

At this stage of development, the system is configured to only work with South African ID cards and books as well as student cards from the University of Pretoria. It is, however, possible to add a template for another type of documentation to the system.

1.1 Requirements and Installation Instructions

The Hutts Verification system can be installed on a user's computer by either making use of Docker or building the source code manually.

1.1.1 Installing via Docker

In order to install the Hutts Verification system by making use of Docker, please make sure that Docker is installed on your system before proceeding with the installation.

Run the following commands in order to get started with Docker:

```
docker pull andreasnel/jhutts:latest
docker run -d --rm -p <your preferred port>:5000 andreasnel/jhutts
```

That's it! You can now make requests to the server running in the docker container exactly like in our section **Using the Server**.

1.1.2 Installing from Source Code

Hutts Verification can also be installed completely from the source code available at https://github.com/javaTheHutts/Java-the-Hutts

NOTE: This installation may take anything from 30 minutes to 3 hours, depending on which requirements have already been installed on the user's computer.

1. Install OpenCV 3.2.0, Python3 and all virtual environments as explained here. Please note that the version in the tutorial is different from the required version (3.2.0).
2. Clone the repository for the source code with `git clone https://github.com/javaTheHutts/Java-the-Hutts.git`
3. Run the following commands after the successful installation and linking of OpenCV to your virtual environment:

```
sudo apt-get install -y libzbar0 libzbar-dev libboost-all-dev tesseract-ocr
```

4. Run the following commands while your OpenCV virtual environment is activated (this installs all the dependencies of the project):

```
pip3 install pybuilder
pip3 install pyzbar==0.1.4 pyzbar[scripts] zbar-py==1.0.4
pip3 install -r requirements.txt
pyb install_dependencies
```

5. Install the project with the following commands:

```
pyb analyze full publish
cd target/dist/Java-the-Hutts-1.0.dev0/
python3 setup.py install
```

6. In order to test whether the installation was successful, the user should be able to execute the `import hutts_verification` statement in any Python 3 shell without error.

1.2 Using the Server

This section deals on how to make requests to the built-in server that a user can run when Hutts Verification has been installed.

All requests to the server should be made by making use of the `POST` method as well as setting the MIME-type of the data to `application/json` (of course, this also implies that the data of the request should be in JSON) in order to correctly send requests to and receive responses from the server.

1.2.1 Extraction

Extract All This request extracts both the textual and the facial data from the ID document.

URL: http://localhost:5000/extractAll.

Sample Data:

```
{
    "idPhoto": "data:image/jpeg;base64,/9j/4QTDRXh......"
}
```

Response:

```
{
    "extracted_face": "data:image/jpg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD...",
    "text_extract_result": {
        "country_of_birth": <string>,
        "date_of_birth": <string>,
        "identity_number": <string>,
        "names": <string>,
        "sex": <string>,
        "status": <string>,
        "surname": <string>
    }
}
```

Extract Text This request only extracts the textual data from the ID document.

URL: http://localhost:5000/extractText.

Sample Data:

```
{
    "idPhoto": "data:image/jpeg;base64,/9j/4QTDRXh..."
}
```

Response:

```
{
    "country_of_birth": <string>,
    "date_of_birth": <string>,
    "identity_number": <string>,
    "names": <string>,
    "sex": <string>,
    "status": <string>,
    "surname": <string>
}
```

Extract Face This request only extracts the image of the person's face from the ID document.

URL: http://localhost:5000/extractFace.

Sample Data:

```
{
    "idPhoto": "data:image/jpeg;base64,/9j/4QTDRXh..."
}
```

Response:

```
{
    "extracted_face": "data:image/jpg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD..."
}
```

1.2.2 Verification

Verify ID This request verifies an image (containing a face) and provided textual information against the information that can be extracted from an image of the relevant identification documentation (which should also be provided).4

URL: http://localhost:5000/verifyID.

Sample Data:

```
{
    cob: <country of birth string>,
    dob: <date of birth string>,
    face_img: "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD...",
    gender: <gender string>,
    idNumber: <ID number string>,
    id_img: "data: image/jpeg;base64,/9j/4QTDRXhpZgAA...",
    names: <name string>,
    nationality: <nationality string>,
    status: <citizenship status string>,
    surname: <surname string>,
    verbose_verify: false,
    verification_threshold: 75
}
```

Response:

```
{
    "face_match": 84.15386465700645,
    "is_match": true,
    "is_pass": false,
    "text_match": 14.29,
    "total_match": 56.20831879420387
}
```

1.3 hutts_verification

1.3.1 hutts_verification package

Subpackages

hutts_verification.id_contexts package

Submodules

hutts_verification.id_contexts.id_context module

This file contains the abstraction of all ID contexts, which contain the necessary information and settings specific to a particular ID document type.

class FieldType

Bases: `enum.Enum`

An enumerator used to specify the field type for extracted ID information.

DATE_HYPHENATED = 3

MIXED = 2

NUMERIC_ONLY = 1

TEXT_ONLY = 0

class IDContext (*match_contexts*)

Bases: `abc.ABC`

This class is an abstraction of all ID contexts. The ID contexts serve to contain the necessary information and settings specific to a particular ID document type for operations such as extracting information from an ID OCR string.

match_contexts (list) A list of dictionaries that contain the contextual information used in the process of retrieving field values from the OCR output string. Example:

```
{
    'field': 'surname',           # The field name - can be set␣
↪to any string one desires.
    'find': 'surname',            # A string to be used for␣
↪matching field names.
                                  # in the OCR output string␣
↪(used to know what to look for).
    'field_type':                 # Indicates if the field␣
↪value is to be treated as alphanumeric or
       FieldType.TEXT_ONLY        # just numeric or just␣
↪alphabetical characters.
                                  # (e.g. indicates that all␣
↪numbers from field value should be removed
                                  # if the field type is TEXT_
↪ONLY).
    'line_type': TITLED_NEWLINE   # Indicates the type of line␣
↪to be considered when looking for the
                                  # field value relative to the
↪'find' value.
                                  # (e.g. TITLED_NEWLINE␣
↪indicates that the field value is preceded
                                  # by a field name/title and a␣
↪newline).
    'multi_line': True,           # Indicates that the field␣
↪value spans multiple lines.
    'multi_line_end': 'names'     # (Optional, unless multi_
↪line is true) A string identifying the next
                                  # field name that indicates␣
↪the end of the multi-line field value.
    'to_uppercase': False,        # (Optional) Indicates that␣
↪the retrieved field value must be
                                  # converted to uppercase.
}
```

get_id_info(*id_string*, *barcode_data=None*, *ignore_fields=None*, *fuzzy_min_ratio=60.0*, *max_multi_line=2*)

Responsible for filtering undesirable fields to be retrieved as well as delegating the responsibility of extracting ID information from an OCR string and housing said information in a convenient dictionary. Some type checking is done to reduce the likelihood of errors further down the call stack.

Parameters

- **id_string** – A string containing some ID information.
- **barcode_data** – A dictionary object containing information extracted from a barcode.
- **ignore_fields** – A list containing fields which are to be ignored during extraction.
- **fuzzy_min_ratio** – The threshold ratio for a minimum, acceptable ratio of fuzziness when comparing two strings.
- **max_multi_line** – Specifies the maximum number of lines that is to be extracted from fields that are noted as running onto multiple lines.

Returns:

- (dict): A dictionary object containing the relevant, extracted ID information.

Raises:

- TypeError: If id_string is not a string.
- TypeError: If barcode_data is not a dictionary.

class LineType

Bases: `enum.Enum`

An enumerator used to specify the line type for extracted ID information.

TITLED_ADJACENT = 1

TITLED_NEWLINE = 0

UNTITLED_ADJACENT = 3

UNTITLED_NEWLINE = 2

hutts_verification.id_contexts.sa_id module

This file contains the abstraction and high-level logic of South African ID contexts.

class SAID(*match_contexts*)

Bases: *hutts_verification.id_contexts.id_context.IDContext*

An abstract class for South African IDs. Contains the high-level logic that is relevant to all South African IDs.

MIN_AGE_DELTA = 15

POST_PROCESS_MIN_FUZZY_RATIO = 70.0

VALID_ID_LENGTH = 13

YEAR_DELTA = 100

validate_id_number(*id_number*)

Determines whether a given id number is valid or not.

Parameters (str) (*id_number*) – An ID number that is to be validated.

Returns:

- (bool): True if the id number is valid, False otherwise.

Raises:

- TypeError: If id_number is not a string containing only numeric characters.

hutts_verification.id_contexts.sa_id_book module

This file contains the logic for South African ID book (post 1994) context.

class SAIDBook

Bases: *hutts_verification.id_contexts.sa_id.SAID*

A class that represents an ID context for a South African ID book (post 1994). It supplies some of the concrete information, such as the match contexts, to the classes higher up in inheritance hierarchy and implements abstract methods defined by its parent.

hutts_verification.id_contexts.sa_id_book_old module

This file contains the logic for South African ID book (pre 1994) context.

class SAIDBookOld

Bases: *hutts_verification.id_contexts.sa_id.SAID*

A class that represents an ID context for a South African ID book (pre 1994). It supplies some of the concrete information, such as the match contexts, to the classes higher up in inheritance hierarchy and implements abstract methods defined by its parent.

hutts_verification.id_contexts.sa_id_card module

This file contains the logic for South African ID card context.

class SAIDCard

Bases: *hutts_verification.id_contexts.sa_id.SAID*

A class that represents an ID context for a South African ID card. It supplies some of the concrete information, such as the match contexts, to the classes higher up in inheritance hierarchy and implements abstract methods defined by its parent.

hutts_verification.id_contexts.up_student_card module

This file contains the logic for University of Pretoria ID card context. It is mainly intended for demonstration purposes.

class UPStudentCard

Bases: *hutts_verification.id_contexts.id_context.IDContext*

A class that represents an ID context for a University of Pretoria ID card.

Module contents

hutts_verification.image_preprocessing package

Submodules

hutts_verification.image_preprocessing.blur_manager module

Wraps all the functionality required for applying blurring techniques to an image.

class BlurManager(*blur_type*, *kernel_size*)

Bases: `object`

The blur is responsible for applying different blur techniques to the images passed.

apply(*image*)

This performs the blurring.

Parameters **image** – The image to be blurred.

Raises:

- NameError: If invalid blur type is provided i.e. Normal, Gaussian or Median.

Returns:

- (obj): The blurred OpenCV image.

blur(*image, blur_kernel=[(3, 3)]*)

This function applies basic blurring to the image passed.

Parameters

- **(obj)** (*image*) – OpenCV image to which basic blurring should be applied to.
- **(int list)** (*blur_kernel*) – Represent the kernel dimension by which basic blurring should be applied to.

Raises:

- ValueError: If a blur_kernel with an invalid length is provided.
- TypeError: If a blur_kernel is not of type list.

Returns:

- (obj): A modified copy of the OpenCV image where basic blurring was applied to the image.

gaussianBlur(*image, blur_kernel=[(7, 7)]*)

This function applies Gaussian blurring to the image passed.

Parameters

- **(obj)** (*image*) – OpenCV image to which Gaussian blurring should be applied to.
- **(Integer list)** (*blur_kernel*) – Represent the kernel dimension by which basic blurring should be applied to.

Raises:

- ValueError: If a blur_kernel with an invalid length is provided.
- TypeError: If a blur_kernel is not of type list.

Returns:

- (obj): A modified copy of the OpenCV image where Gaussian blurring was applied to the image.

medianBlur(*image, blur_kernel=[3]*)

This function applies Median blurring to the image passed.

Parameters

- **(obj)** (*image*) – OpenCV image to which Median blurring should be applied to.
- **(int list)** (*blur_kernel*) – Represent the kernel dimension by which median blurring should be applied to.

Raises:

- TypeError: If a blur_kernel is not of type list.
- ValueError: If a blur_kernel with an invalid length is provided.

Returns: (obj): A modified copy of the image where Median blurring was applied to the image.

hutts_verification.image_preprocessing.build_director module

This class is responsible for controlling the PipelineBuilder.

class BuildDirector

Bases: `object`

The BuildDirector constructs the Pipeline using the PipelineBuilder.

static construct_face_extract_pipeline()

This function constructs the pipeline for face extraction. This includes building different managers with their specific parameters. These managers will be called within the pipeline when executed.

Returns:

- (obj): Pipeline for facial extraction.

static construct_text_extract_pipeline(*preferences*, *identification_type*)

This function constructs the pipeline for text extraction. This includes building different managers with their specific parameters. These managers will be called within the pipeline when executed.

Parameters

- **(dict)** (*preferences*) – User-specified techniques to use in pipeline.
- **(str)** (*identification_type*) – Contains the type of identification, this is used to determine which techniques are used.

Returns:

- (obj): Pipeline used for text extraction.

hutts_verification.image_preprocessing.color_manager module

Wraps all the functionality required to manipulate the color channels in an image.

class ColorManager(*color_extraction_type*, *channel='green'*, *kernel_size=(13, 7)*)

Bases: `object`

The Color manager is responsible for applying several color management techniques to the image passed.

apply(*image*)

This performs the specified processing technique.

Parameters image – The image to which the technique must be applied.

Raises:

- **NameError: If an invalid color extraction type is provided (other than** histogram, extract, black hat or top hat (white hat)).

Returns:

- (obj): The modified OpenCV image.

blackHat(*image*, *rect_kernel_size=(13, 7)*)

This function applies blackhat color changes to the image passed.

Parameters

- **(obj)** (*image*) – OpenCV image to which black hat color changes should be applied to.

- **`(list)`** (*`rect_kernel_size`*) – Represents the kernel dimensions by which blackHat morphology changes should be applied to.

Raises:

- TypeError: If the kernel size type is not a tuple.
- ValueError: If the kernel size tuple contains more than 2 items.

Returns:

- (obj): A modified copy of the image blackHat morphology applied to an image.

`extractChannel`(*image*, *image_channel='green'*)

This function extracts a selected color channel from an image.

Parameters

- **`(obj)`** (*`image`*) – OpenCV image for which the image channel should be removed.
- **`(str)`** (*`image_channel`*) – Color that should be removed (valid colors: 'red', 'green', 'blue').

Raises:

- NameError: If invalid colour is selected (not red, green, blue).

Returns:

- (obj): A copy of the image passed but with a color channel removed.

`static histEqualisation`(*image*)

This function applies histogram equalisation to the image passed.

Parameters **`(obj)`** (*`image`*) – OpenCV image to which histogram equalisation should be applied to.

Returns:

- (obj): The Histogram equalised image.

`topHat`(*image*, *rect_kernel_size=(13, 7)*)

This function applies tophat color changes to the image passed.

Parameters

- **`(obj)`** (*`image`*) – Image to which top hat color changes should be applied to.
- **`(list)`** (*`rect_kernel_size`*) – Represents the kernel dimension by which topHat morphology changes should be applied to.

Raises:

- TypeError: If the kernel size type is not a tuple.
- ValueError: If the kernel size tuple contains more than 2 items.

Returns:

- (obj): A modified copy of the image with topHat morphology applied to it.

hutts_verification.image_preprocessing.face_manager module

Wraps all the functionality necessary for extracting a face from an image.

class FaceDetector(*shape_predictor_path*)

Bases: `object`

The FaceDetector class is responsible for:

1. Detecting the face.
2. Extracting a face from an image.
3. Applying blurring on a detected face in an image.

blur_face(*image*)

This function find the faces and apply a blurring effect on the detected region. After the region has been blurred, the blurred region is reapplied to the original image. Blurring the face is implemented as a method in the attempt to reduce noise when extracting text from the image later in the image pipeline.

Parameters **(obj)** (*image*) – OpenCV image containing the face we need to detect and blur.

Raises:

- ValueError: If no face can be detected.

Returns:

- (obj): A copy of the image with blurring applied to the face in the image.

detect(*image*)

This function detects the face in the image passed. By making use of the dlib HOG feature image_preprocessing and linear classifier for frontal face detection we are able to detect the face with less false-positive results and without a major time penalty. More Information dlib frontal_face detection: http://dlib.net/imaging.html#get_frontal_face_detector

A check will be done to see if a face is present in the image. If a face is not detected in the image the execution should log that the face was not found and continue with execution. This is due to the fact that face detection might not be critical to a function (like with text extraction) and rather be used to increase accuracy.

Parameters **(obj)** (*image*) – OpenCV image containing the face we need to detect.

Raises:

- ValueError: If no face can be detected.

Returns:

- list(int): This list contains the box coordinates for the region in which the face resides.

extract_face(*image*)

This function finds a face in the image passed and is optimised to align the face before being returned.

Parameters **(obj)** (*image*) – Image containing the face we need to detect and extract.

Raises:

- ValueError: If no face can be detected.

Returns:

- (obj): An image of the aligned face.

hutts_verification.image_preprocessing.pipeline module

This is an object that handles the entire process of extracting data from an image from a high-level perspective.

class Pipeline(*blur_manager=None*, *color_manager=None*, *face_detector=None*, *threshold_manager=None*)

Bases: `object`

The Pipeline will perform all necessary processing on the image and is built by the PipelineBuilder.

Parameters

- **(BlurManager)** (`blur_manager`) – The BlurManager that is used in this pipeline.
- **(ColorManager)** (`color_manager`) – The ColorManager that is used in this pipeline.
- **(FaceDetector)** (*face_detector*) – The FaceDetector that is used in this pipeline.
- **(ThresholdManager)** (*threshold_manager*) – The ThresholdManager that is used in this pipeline.

process_face_extraction(*image*)

This function applies all the processing needed to extract a face from a image.

Parameters (obj) (*image*) – Image to which processing should be applied to.

Returns:

- (obj): The processed image.

process_text_extraction(*useIO*, *image*, *remove_face=False*)

This function applies all the processing needed to extract text from a image.

Parameters

- **(boolean)** (*remove_face*) – Whether or not to write images to disk.
- **(obj)** (*image*) – Image that should be processed.
- **(boolean)** – If the remove face flag is set to true, extra processes will be activated during the pre-processing phase to remove the face from the image.

Returns:

- (obj): The processed image.

hutts_verification.image_preprocessing.pipeline_builder module

The class that is responsible for creating a Pipeline. This class is the Builder of the Builder design pattern.

class PipelineBuilder

Bases: `object`

The PipelineBuilder will assemble all the parts of the Pipeline.

get_result()

This function returns the fully-assembled pipeline.

Returns: (obj): The Pipeline.

set_blur_manager(*value*)

This function adds the specified blur manager to the pipeline.

Parameters **(BlurManager)** (*value*) – BlurManager object to be added.

set_color_manager(*value*)
This function adds the specified color manager to the pipeline.

Parameters **(ColorManager)** (*value*) – ColorManager object to be added.

set_face_detector(*value*)
This function adds the specified face detector to the pipeline.

Parameters **(FaceDetector)** (*value*) – FaceDetector object to be added.

set_threshold_manager(*value*)
This function adds the specified threshold manager to the pipeline.

Parameters **(ThresholdManager)** (*value*) – ThresholdManager object to be added.

hutts_verification.image_preprocessing.template_matching module

Wraps the functionality required to dynamically deduce the type of identification documentation that has been provided in an image.

class TemplateMatching
Bases: object

The TemplateMatching class receives template images to identify the type of identification that is used in the image. Thus you provide it with templates and it will identify whether you used an ID card, ID book etc.

identify(*source*)
This function identifies the src image by searching for the templates provided.

:param source (obj) : The image that needs to be identified.

Returns:

- (str) : Either type of the image or None if the type could not be identified.

hutts_verification.image_preprocessing.thresholding_manager module

Wraps all the functions related to applying thresholding techniques to an image.

class ThresholdingManager(*thresholding_type*)
Bases: object

The Thresholding manager is responsible for applying the different types of thresholding techniques.

static adaptiveThresholding(*image*)
This function applies a simple adaptive thresholding to the image passed.

Parameters **(obj)** (*image*) – Image to which thresholding should be applied.

Raises:

- TypeError: If a parameter is passed that is not of type Numpy array.

Returns:

- (obj): The image, with adaptive thresholding applied to it.

apply(*image*)
This performs the thresholding based on the predefined technique.

Parameters **`image`** – The image to which the thresholding must be applied.

Raises:

- NameError: If a thresholding type other than ‘adaptive’ or ‘otsu’ is provided.

Returns:

- (obj): The image, with the appropriate thresholding applied to it.

`static otsuThresholding`(*image*)

This function applies a simple Binary Inverse Otsu thresholding to the image passed.

Parameters **`(obj)`** (*`image`*) – Image to which thresholding should be applied.

Raises:

- TypeError: If a parameter is passed that is not of type Numpy array.

Returns:

- (obj): The image, with otsu thresholding applied to it.

Module contents

hutts_verification.image_processing package

Submodules

hutts_verification.image_processing.barcode_manager module

A class that is responsible for detecting a barcode and extracting information from said barcode.

`class BarCodeManager`

Bases: `object`

The BarCode Manager is responsible for:

1. Detecting the barcode.
2. Extracting any information on the detected barcode.
3. Applying blurring to the detected barcode to reduce noise.

`apply_barcode_blur`(*image*, *box*, *dilation_intensity=2*)

This function applies blurring to a detected barcode region to reduce noise in the image. The barcode region is first extracted, then blurring is applied, after blurring is applied the blurred out barcode is reapplied to the original image.

Parameters

- **`(obj)`** (*`image`*) – Image containing a Barcode.
- **`(int list)`** (*`box`*) – The box is an integer list containing the box region coordinates of the barcode location.
- **`(int)`** (*`dilation_intensity`*) – Indicates the intensity by which the barcode should be dilated. The greater the intensity the greater the extent of dilation.

Returns: (obj): The original image with blurring applied to the barcode region in the image.

detect(*image*)

This function detects a region containing a barcode if a barcode is present in the image passed. Barcodes supported:

- EAN
- UPC
- Code 39
- Code 93
- Code 128
- ITF

For more information on Barcode types: https://www.scandit.com/types-barcodes-choosing-right-barcode/.

Parameters (obj) (*image*) – Image containing the potential barcode.

Returns:

- (boolean): Indicates whether a barcode was detected or not.
- (obj): Return the detected barcode or the original image, based on whether the barcode was found.
- (int list): This list contains the box coordinates for the region in which the barcode resides.

Raises:

- TypeError: If a none numpy array value has been passed.

get_barcode_info(*image*)

This function returns scanned barcode information.

Parameters (obj) (*image*) – Image containing a Barcode.

Returns:

- (boolean): Whether the function was able to extract information from the barcode.
- (str): A UTF-8 String of the data extracted from the barcode (empty if nothing was extracted).
- (obj): A copy of the original image.

hutts_verification.image_processing.context_manager module

This file contains the logic used to manage the various ID contexts. Should ideally be extended to do more than it currently does.

class ContextManager

Bases: `object`

A class responsible for managing and maintaining the various ID contexts.

_sa_id_card (SAIDCard) A South African ID card context.

_sa_id_book (SAIDBook) A South African ID book context.

_sa_id_book_old (SAIDBookOld) An old South African ID book context.

_up_card (UPStudentCard) A University of Pretoria staff/student card context.

get_id_context(*id_type*)
Returns an ID context based on the ID type that is passed in as an arg.

Parameters **(str)** (*id_type*) – A string indicating a type of ID.

Returns:

- (IDContext): An IDContext object determined by the ID type passed in as an arg.
- (None): In the event that the ID type is unrecognisable.

hutts_verification.image_processing.controllers module

Handles all requests relevant to the extraction service of the API.

extract_all()
Sample function to extract face and text from image received.

URL: http://localhost:5000/extractAll.

extract_face()
Sample function to extract face from image received.

URL: http://localhost:5000/extractFace.

extract_text()
Sample function to extract text from image received.

URL: http://localhost:5000/extractText.

face_extraction_response(*use_io*, *image*, *text_extract_result=None*)
This function converts the extracted cv2 image and converts it to a jpg image. Furthermore, the jpg image is converted to Base64 jpg type and returned. If text extraction results are provided the response will contain the data of text extraction result as well.

Parameters

- **(boolean)** (*use_io*) – Whether or not images should be written to disk.
- **(obj)** (*image*) – The cv2 (numpy) image that should be converted to jpg.
- **(dict)** (*text_extract_result*) – The extracted text results.

Returns:

- (obj): The response object that contains the information for HTTP transmission.

hutts_verification.image_processing.sample_extract module

This class wraps all the functionality required to extract text from an image.

class FaceExtractor
Bases: `object`

The FaceExtractor extracts the face region for the image passed.

extract(*img*, *use_io*)
This function is a sample that demonstrates how the face would be extracted.

Parameters

- **(obj)** (*img*) – The image of the ID that contains the face that must be extracted.

- **(boolean)** (*use_io*) – Whether or not images should be written to disk.

Returns:

- (obj): The extracted and aligned facial image.

class TextExtractor(*preferences*)

Bases: `object`

The TextExtractor extracts text from the ID image.

extract(*img*)

This function is a sample that demonstrates how text would be extracted.

Parameters **(obj)** (*img*) – The image of the ID that contains the text to be extracted.

Returns:

- id_details (obj): The extracted information.

hutts_verification.image_processing.simplification_manager module

Wraps functions that are used to simplify or ease the image processing process.

class SimplificationManager

Bases: `object`

The Simplification manger is used to remove unwanted content in an image thus simplifying process like OCR and facial comparisons.

perspectiveTransformation(*image*, *use_io*)

The perspective transformation takes the image passed and applies edge detection and a function to detect the contours of a identification document. If contours of an identification document is detected the image is converted from a non-perspective view to a perspective view.

Parameters

- **(obj)** (*image*) – Image containing a identification document.
- **(boolean)** (*use_io*) – Whether or not to write images to disk.

Returns:

- (obj): The transformed image.

Raises:

- TypeError: If a parameter is passed that is not of type Numpy array.

hutts_verification.image_processing.text_cleaner module

This file contains the logic used to manage the removal of characters from an input string.

class TextCleaner

Bases: `object`

This class is encapsulates the logic required to clean the OCR output string produced from an image of an ID.

_deplorables (list) A list of strings that contain characters that is to be filtered out from the OCR output string during string cleaning.

clean_up(*in_string*, *deplorables=None*)
This function serves to receive an input string, clean it up through removing undesirable characters and unnecessary whitespace, and to return the cleaned string.

Parameters

- **(str)** (*in_string*) – The input string that is to be cleaned.
- **(list)** (*deplorables*) – A list of characters that are to be filtered from the input string.

Returns:

- (str): A string that has been stripped of undesirable characters and unnecessary whitespace.

Raises:

- TypeError: If in_string is not a string.
- TypeError: If deplorables is not a list of strings.

Module contents

hutts_verification.utils package

Submodules

hutts_verification.utils.hutts_logger module

Wraps the logic required to setup and use a custom logger while disabling the built-in flask logger.

Example usage:

First import the logger from the hutts_logger module… from hutts_verification.utils.hutts_logger import logger then start logging.

`logger.info('logging an example')`

There are 5 logging levels, as shown below with their corresponding function call (from lowest to highest level):

- DEBUG - `logger.debug(message)`
- INFO - `logger.info(message)`
- WARNING - `logger.warning(message)`
- ERROR - `logger.error(message)`
- CRITICAL - `logger.critical(message)`

The default level of the logger will be INFO, unless the flask app is run in debug mode, in which case the logger level will be DEBUG. What this means is that messages lower than INFO, i.e. DEBUG, will not be shown, again this is unless the flask app is run in debug mode.

LOGGING_DEFAULT_LEVEL = 10
Specifies the date format to be used by the various logging formatters.

LOGGING_LOGGER_NAME = 'hutts_logger'
Specifies the default level for logging.

LOGGING_LOG_DATE_FMT = '%Y-%m-%d %H:%M:%S'
Indicates whether or not to log to console.

`LOGGING_LOG_TO_CONSOLE = True`
Specifies the log format to be used when logging to the console.

`LOGGING_LOG_TO_CONSOLE_COLOURS = {'DEBUG': 'bold_cyan', 'ERROR': 'bold_red', 'CRITICAL': 'b`
Specifies the secondary colours to be used to indicate the different levels when logging to console. For a more detailed description see the colorlog documentation: https://github.com/borntyping/python-colorlog.

`LOGGING_LOG_TO_CONSOLE_FMT = '[%(asctime)s.%(msecs)03d]%(log_color)s[%(levelname)8s]%(rese`
Specifies the colours to be used to indicate the different levels when logging to console. For a more detailed description see the colorlog documentation: https://github.com/borntyping/python-colorlog.

`LOGGING_LOG_TO_CONSOLE_SEC_COLOURS = {'message': {'ERROR': 'red', 'CRITICAL': 'bold_red,b`
Indicates whether or not to log to a file.

`LOGGING_LOG_TO_FILE = True`
Specifies the name of the log file.

`LOGGING_LOG_TO_FILE_BACKUP_COUNT = 1`
Specifies the encoding for the log file.

`LOGGING_LOG_TO_FILE_DEFAULT_DIR = 'log/'`
Specifies the log format to be used when logging to a file.

`LOGGING_LOG_TO_FILE_ENCODING = 'utf8'`
A global reference to the custom logger to be used. It is initialised to the default python logger to avoid errors during installation of packages.

`LOGGING_LOG_TO_FILE_FILENAME = 'hutts_verification.log'`
Specifies the default directory for the log file in the event that one is not specified during setup.

`LOGGING_LOG_TO_FILE_FMT = '[%(asctime)s.%(msecs)03d][%(levelname)8s] -- (%(filename)s:%(li`
Specifies the maximum number of bytes for the log file before a rotate (assuming a rotating file is used).

`LOGGING_LOG_TO_FILE_MAX_BYTES = 100000`
Specifies the number of backups for the log file (assuming a rotating file is used).

`disable_flask_logging`(*app_instance*)
This function disables the flask logging, which interferes with the custom logger.

Parameters **`(obj)`** (*`app_instance`*) – A reference to the current flask server application.

`get_console_handler`()
This function is responsible for creating a console log handler with a global format and returning it.

Returns:

- (obj): A log handler that is responsible for logging to the console.

`get_file_handler`(*log_dir=None*)
This function is responsible for creating a file log handler with a global format and returning it.

Returns:

- (obj): A log handler that is responsible for logging to a file.

`prettify_json_message`(*json_message*)
This function is a helper function that is used to prettify a json/dict message obj so that is more readable for humans when it is logged.

Parameters **`(dict)`** (*`json_message`*) – A message that is to be prettified before being logged.

Returns:

- (str): A prettified json message string.

setup_logger()
This function is responsible for creating the custom logger and delegating the creation of its handlers.

hutts_verification.utils.image_handling module

Utility functions to manage image handling from given parameters.

grab_image(*path=None*, *stream=None*, *url=None*, *string=None*)
This function grabs the image from URL, or image path and applies necessary changes to the grabbed images so that the image is compatible with OpenCV operation.

Parameters

- **(path)** (*str*) – The path to the image if it resides on disk.
- **(stream)** (*str*) – A stream of text representing an image upload.
- **(url)** (*str*) – URL representing a path to where an image should be fetched.
- **(string)** (*str*) – A base64 encoded string of the image.

Raises:

- ValueError: If no path, stream, URL or Base64 string was found.

Returns:

- (obj): Image that is now compatible with OpenCV operations.

hutts_verification.utils.pypath module

Utilities to help solve Python path/directory/file issues regarding files being installed in either the dist-packages or the site-packages folder.

correct_path(*path*)
This function checks if the given path exists in most known Python package installation directories and returns the corresponding path.

- path (str || Path): The path that has to be checked.

Returns: (str): The correct path if it exists, else None.

Module contents

hutts_verification.verification package

Submodules

hutts_verification.verification.controllers module

Handles all requests relevant to the verification service of the API.

manage_text_extractor(*image_of_id*)
This function manages the text extraction from an ID images. Management includes preparing text extraction preferences and receiving extracted text.

Parameters **(obj)** (*image_of_id*) – An image of an ID that text must be extracted from.

Returns:

- preferences (dict): Prepared list of preferences. May contain additional text extraction or logger preferences.
- extracted_text (json object): A collection of text extracted from the ID.

manage_text_verification(*preferences*, *extracted_text*, *entered_details*)

The function manages the preparation before text verification and result of the text verification it self Management includes preparing logger functionality and controlling match percentages and messages.

Parameters

- **(list)** (*preferences*) – A list of preferences containing details about logger functionality.
- **(JSON object)** (*extracted_text*) – Contains text extracted from ID image.
- **(dict)** (*entered_details*) – Dictionary containing information that needs to be verified.

Returns:

- (float): Value representing the accuracy with which the entered details matches that of the extracted text.
- (dict): Contains individual match percentages for different fields.
- (boolean): Indicates if text_verification passes based on the threshold value.

match_faces(*image_of_id*, *face*)

This function receives two images that receive images of faces that need to be verified. It is expected that an image of an ID and an image of a Profile picture will be received. Even if the expected images are not received the function will still apply a best effort solution.

Parameters

- **(obj)** (*face*) – An image of an ID that contains a face that needs to be verified.
- **(obj)** – A image of a face that needs to be verified.

Returns:

- boolean: Whether the two faces match (the distance between them is above the threshold value).
- float: Return Euclidean distance between the vector representations of the two faces.

receive_details()

This function receives the details that need to be verified from the flask handler.

Returns:

- (dict): Details that need to be verified with that extracted from image.

receive_faces(*match_face=True*)

This function receives faces/ID from request flask handler. The function checks for multiple means of receiving the faces/ID. These include

- Receiving image by file path
- Receiving image by URL

- Receiving image by file Stream

It is expected that an image of a face and an image of an ID will be sent. However, if the order is not followed that system will still be able to return the best effort result without loss of accuracy.

Parameters **(boolean)** (*match_face*) – Indicates if an additional profile of a face should be extracted. If an additional face should not be extracted simply return the ID image.

Returns:

- (obj): An image of a ID.
- (obj): An image of a face if match_face is set to True.

verify_faces()

Sample function to return a match percentage of an ID face image and picture of face.

URL: http://localhost:5000/verifyFaces.

verify_id()

Sample function to return a match percentage of an ID image and provided personal information and picture of face.

URL: http://localhost:5000/verifyID.

verify_info()

Sample function to return a match percentage of an ID image and provided personal information.

hutts_verification.verification.face_verify module

A class that is used to extract and compare two images of faces and calculate the resulting match.

class FaceVerify(*shape_predictor_path*, *face_recognition_path*)

Bases: `object`

The FaceVerify class is responsible for

1. Detecting the face.
2. Generating a threshold value that determines the likeness of two individuals in an image.

verify(*face1*, *face2*, *threshold=0.55*)

This function determines a percentage value of how close the faces in the images passed are to each other if the determined value if below the threshold value passed by the user a boolean value of True is returned indicating that the faces in the images passed indeed match.

The verify function makes use of the dlib library which guarantees 99.38% accuracy on the standard Labeled Faces in the Wild benchmark.

Parameters

- **(obj)** (*face2*) – The first image containing the face that should be compared.
- **(obj)** – The second image containing the face that should be compared.
- **(float)** (*threshold*) – The threshold value determines at what distance the two images are considered the same. If a verify score is below the threshold value the faces are considered a match. The Labled Faces in the Wild benchmark recommend a default threshold of 0.6 but a threshold of 0.55 was decided on to ensure higher confidence in results.

Returns:

- (boolean): Represents if two face indeed match.
- (float): The Euclidean distance between the vector representations of the two faces.

Raises:

- ValueError: If no face can be detected then no faces can be matched and the operation should be aborted.

hutts_verification.verification.text_verify module

Contains the logic used to verify the extracted text from a form of ID.

class TextVerify

Bases: `object`

This class is responsible for the verification of text that is extracted from an ID.

validate_id_number(*id_number*, *valid_length=13*)

Determines whether a given id number is valid or not.

Parameters

- **(str)** (*id_number*) – The ID number that has to be validated.
- **(int)** (*valid_length*) – Specifies the length of a given id number to be considered as valid.

Returns:

- (boolean): True if the id number is valid, False otherwise.

Raises: TypeError: If id_number is not a string containing only numeric characters. TypeError: If valid_length is not an integer.

verify(*extracted*, *verifier*, *threshold=75.0*, *min_matches=4*, *verbose=False*)

This function is responsible for the verification of text that is extracted from an ID and is passed in, along with information that is to be used to verify the extracted text.

Parameters

- **(dict)** (*verifier*) – The information that was extracted from an ID.
- **(dict)** – The information against which the extracted data is to be verified.
- **(float)** (*threshold*) – A percentage used to determine if the match percentage is accepted as verified.
- **(int)** (*min_matches*) – The minimum number of matches needed to allow a positive match.
- **(boolean)** (*verbose*) – Indicates whether or not to return all of the calculated match percentages.

Returns:

- **((bool, float) | dict): The first value returned is a bool that indicates whether or not the total** percentage match is above the specified threshold value, while the second return value is the total percentage match value if verbose is False, or returns a dict of all the determined percentage match values if verbose is True.

Raises: TypeError: If any parameter is not of the correct type.

Module contents

Module contents

1.4 Frequently Asked Questions

1.4.1 Can Hutts Verification extract data from a passport?

No, at this stage in development the system can only extract data from a South African ID card, ID book and student card from the University of Pretoria. We do plan on adding this feature in the future, however.

1.4.2 Can Hutts Verification extract data from a driver's license?

No, at this stage in development the system can only extract data from a South African ID card, ID book and student card from the University of Pretoria.

A South African driver's license is especially difficult to work with, due to the watermarks present on the lamination and the extremely poor quality font that is used on the document.

1.4.3 Why does some processes take longer than others?

This can be due to a number of reasons. They may include:

- The larger the picture that is uploaded, the longer it takes the system to process the image.
- The system cannot identify the documentation type and cannot work with optimized parameters, therefore making use of default processing mechanisms.
- The network speed is slow. This affects the uploading and downloading of files.
- If you are running the server on your local computer, the hardware that is present on your computer has a large impact on the maximum potential of the system.
- The busier the background, the more data the system has to scan through in order to extract the correct data, which can lead to speed issues.
- Lower quality images also tend to take longer than higher quality images, due to the system having to try and take a best guess effort to determine what kind of data is extracted.

1.4.4 Why is the Docker image so large?

This is due to the large number of libraries that are used to process an image in the Hutts Verification system. OpenCV, which is the backbone of our system, is already a very large installation inside its own Docker container. The Hutts Verification container builds on top of that container, and therefore the size of the final Docker image is extremely large due to the large number of dependencies.

1.4.5 Why do some types of documentation work better than others?

Each type of documentation looks different to another. This means that some types of documentation are easier to process than others, leading to a speed increase.

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX