**Natural Language Processing for PDF/TIFF/Image Documents  
Computer Vision for Image Data  
Users Guide  
High Precision Natural Language Processing for PDF/TIFF/Image Documents and Computer Vision for Images  
Users Guide, Gap v0.9**

# 1. Introduction

The target audience for this users guide are your software developers whom will be integrating the core inner block into your product and/or service. It is not meant to be a complete reference guide or comprehensive tutorial, but a brief get started guide.

To utilize this module you must have installed:

1. This Python module.
2. Python 3.6 or latter
3. Ghostscript ©(open source from Artifex).
4. Tesseract ©(open source from Google).
5. Magick ©(open source from Image Magic).
6. NLTK Toolkit (open source).
7. Unidecode (open source).
8. HD5 (open source)
9. Numpy (open source)

# 2. SPLITTER Module

## 2.1 Document Loading

To load a PDF document, TIFF facsimile or image captured document you create a Document (class) object, passing as parameters the path to the PDF/TIFF/image document and a path for storing the split pages/text. Below is a code example.

**from** splitter **import** Document, Page  
document = Document(“yourdocument.pdf”, “storage\_path”)

## 2.2 Page Splitting

## Upon instantiating a document object, the corresponding PDF document or TIFF facsimile is automatically split into the corresponding PDF or TIFF pages, utilizing Ghostscript (PDF) and Magick (TIFF). Each PDF/TIFF page will be stored separately in the storage path with the following naming convention:

<document basename><pageno>.<suffix> , where <suffix> is either *pdf* or *tif*

The module automatically detects if a PDF document is a digital (text) or scanned PDF (image). For digital documents, the text is extracted directly from the PDF page using Ghostscript and stored separately in the storage path with the following naming convention:

<document basename><pageno>.txt

2.3 OCR  
  
If the document is a scanned PDF, each page image will be extracted using Ghostscript, then OCR using Tesseract to extract the text content from the page image. The page image and corresponding page text are stored separately in the storage path with the following naming convention:

<document basename><pageno>.png  
<document basename><pageno>.txt

If the document is a TIFF facsimile, each page image will be extracted using Magick, then OCR using Tesseract to extract the text content from the page image. The page image and corresponding page text are stored separately in the storage path with the following naming convention:

<document basename><pageno>.tif  
<document basename><pageno>.txt

If the document is an image capture (e.g., JPG), the image is OCR using Tesseract to extract the text content from the page image. The page image and corresponding page text are stored separately in the storage path with the following naming convention:

<document basename><pageno>.<suffix> , where <suffix> is *png* or *jpg*  
<document basename><pageno>.txt

## 2.4 Image Resolution for OCR

The resolution of the image rendered by Ghostscript from a scanned PDF page will affect the OCR quality and processing time. By default the resolution is set to 300. The resolution can be set for a (or all) documents with the static member RESOLUTION of the Document class. This property only affects the rendering of scanned PDF; it does not affect TIFF facsimile or image capture.

# Set the Resolution of Image Extraction of all scanned PDF pages  
Document.RESOLUTION = 150

# Image Extraction and OCR will be done at 150 dpi for all subsequent documents  
document = Document(“scanneddocument.pdf”, “storage\_path”)

## 2.5 Page Access

Each page is represented by a Page (class) object. Access to the page object is obtained from the pages property member of the Document object. The number of pages in the document is returned by the len() builtin operator for the Document class.

document = Document(“yourdocument.pdf”, “storage\_path”)  
   
# Get the number of pages in the PDF document  
npages = len(document)

# Get the page table  
pages = document.pages

# Get the first page  
page1 = pages[0]

# or alternately  
page1 = document[0]

# full path location of the PDF/TIFF or image capture page in storage  
page1\_path = page1.path

## 2.6 Adding Pages

Additional pages can be added to the end of an existing Document object using the += (overridden) operator, where the new page will be fully processed.   
  
 document = Document(“1page.pdf”)

# This will print 1 for 1 page  
print(len(document))

# Create a Page object for an existing PDF page  
new\_page = Page(“page\_to\_add.pdf”)

# Add the page to the end of the document.  
document += new\_page

# This will print 2 showing now that it is a 2 page document.  
print(len(document))

## 2.7 Text Extraction

The raw text for the page is obtained by the text property of the page class. The byte size of the raw text is obtained from the size() method of the page class.

# Get the page table  
pages = document.pages

# Get the first page  
page1 = pages[0]

# Get the total byte size of the raw text  
 bytes = page1.size()

# Get the raw text for the page  
 text = page1.text

The property scanned is set to True if the text was extracted using OCR; otherwise it is false (i.e., origin was digital text).

# Determine if text extraction was obtained by OCR  
scanned = document.scanned

## 2.8 Asynchronous Processing

To enhance concurrent execution between a main thread and worker activities, the Document class supports asynchronous processing of the document (i.e., Page Splitting, OCR and Text Extraction). Asynchronous processing will occur if the optional parameter ehandler is set when instantiating the Document object. Upon completion of the processing, the ehandler is called, where the Document object is passed as a parameter.

**def** done(d):  
 “”” Event Handler for when processing of document is completed “””  
 print(“DONE”, d.document)

# Process the document asynchronously  
document = Document(“yourdocument.pdf”, “storage\_path”, ehandler=done)

## 2.9 NLP Preprocessing of the Text

NLP preprocessing of the text requires the SYNTAX module. The processing of the raw text into NLP sequenced tokens (syntax) is deferred and is executed in a JIT (Just in Time) principle. If installed, the NLP sequenced tokens are access through the words property of the Page class. The first time the property is accessed for a page, the raw text is preprocessed, and then retained in memory for subsequent access.

# Get the page table  
pages = document.pages

# Get the first page  
page1 = pages[0]

# Get the NLP preprocessed text  
words = page1.words

The NLP preprocessed text is stored separately in the storage path with the following naming convention:

<document basename><pageno>.json

## 2.10 NLP Preprocessing Settings (Config)

NLP Preprocessing of the text may be configured for several settings when instantiating a Document object with the optional config parameter, which consists of a list of one or more predefined options.

document = Document(“yourdocument.pdf”, “storage\_path”, config=[options])

# options:

bare # do bare tokenization  
stem = internal | # use builtin stemmer  
 porter | # use NLTK Porter stemmer  
 snowball | # use NLTK Snowball stemmer  
 lancaster | # use NLTK Lancaster stemmer  
 lemma | # use NLTK WordNet lemmatizer  
 nostem # no stemming

pos # Tag each word with NLTK parts of speech  
roman # Romanize latin-1 character encodings into ASCII

## 2.11 Document Reloading

## Once a Document object has been stored, it can later be retrieved from storage, reconstructing the Page and corresponding Words objects. A document object is first instantiated, and then the load() method is called specifying the document name and corresponding storage path. The document name and storage path are used to identify and locate the corresponding stored pages.

# Instantiate a Document object  
document = Document()

# Reload the document’s pages from storage  
document.load( “mydoc.pdf”, “mystorage” )

# This will reload pages whose filenames in the storage match the sequence:  
# mystorage/mydoc1.json  
# mystorage/mydoc2.json  
# ….

## 2.13 Word Frequency Distributions

The distribution of word occurrences and percentage in a document and individual pages are obtained using the properties: bagOfWords, freqDist and termFreq.

The bagOfWords property returns an unordered dictionary of each unique word in the document (or page) as a key, and the number of occurrences as the value.

# Get the bag of words for the document  
 bow = document.bagOfWords  
 print(bow)  
 # will output:  
 # { ‘<word>’: <no. of occurrences>, ‘<word>’: <no. of occurrences>, … }  
 # e.g., { ‘plan’: 20, ‘medical’: 31, ‘doctor’: 2, … }

# Get the bag of words for each page in the document  
 **for** page **in** document.pages:  
 bow = page.bagOfWords

The freqDist property returns a sorted list of each unique word in the document (or page), as a tuple of the word and number of occurrences, sorted by the number of occurrences in descending order.

# Get the word frequency (count) distribution for the document  
 count = document.freqDist  
 print(count)  
 # will output:  
 # [ (‘<word>’, <no. of occurrences>), (‘<word>’: <no. of occurrences>), … ]   
 # e.g., [ (‘medical’, 31), (‘plan’, 20), …, (‘doctor’, 2), … ]

# Get the word frequency distribution for each page in the document  
 **for** page **in** document.pages:  
 count = page.freqDist

The termFreq property returns a sorted list of each unique word in the document (or page), as a tuple of the word and the percentage it occurs in the document, sorted by the percentage in descending order.

# Get the term frequency (TF) distribution for the document  
 tf = document.freqDist  
 print(tf)  
 # will output:   
 # [ (‘<word>’, <percent>), (‘<word>’: <percent>), … ]   
 # e.g., [ (‘medical’, 0.02), (‘plan’, 0.015), … ]

## 2.13 Document and Page Classification

Semantic Classification (e.g., category) of the document and individual pages requires the CLASSIFICATION module. The classification is deferred and is executed in a JIT (Just in Time) principle. If installed, the classification is access through the classification property of the document and page classes, respectively. The first time the property is accessed for a document or page, the NLP sequenced tokens for each page are processed for classification of the content of individual pages and the first page is further processed for the classification of the content of the entire document.

# Get the classification for the document  
document\_classification = document.label

# Get the classification for each page  
**for** page **in** document.pages:  
 classification = page.label

# 3. SYNTAX Module

## 3.1 NLP Processing

The Words (class) object does the NLP preprocessing of the extracted (raw) text. If the extracted text is from a Page object (see SPLITTER), the NLP preprocessing occurs the first time the words property of the Page object is accessed.

**from** syntax **import** Words, Vocabulary

# Get the first page in the document  
page = document.pages[0]

# Get the raw text from the page as a string  
text = page.text

# Get the NLP processed words (Words class) object from the page as a list.  
words = page.words

# Print the object type of words => <class ‘Document.Words’>  
type(words)

## 3.2 Words Properties

The Words (class) object has four public properties: text , words, bagOfWords and freqDist. The text property is used to access the raw text and the words property is used to access the NLP processed tokens from the raw text.

# Get the NLP processed words (Words class) object from the page as a list.  
words = page.words

# Get the original (raw) text as a string  
 text = words.text

# Get the NLP processed words from the original text as a Python list.  
words = words.words

# Print the object type of words => <class ‘list’>  
type(words)

The bagOfWords and freqDist properties are explained later in the guide.

## 3.3 Vocabulary Dictionary

The words property returns a sequenced Python list of words as a dictionary from the Vocabulary class. Each word in the list is of the dictionary format:

{ ‘word’ : word, # The stemmed version of the word  
 ‘lemma’: word, # The lemma version of the word  
 ‘tag’ : tag # The word classification  
 }

## 3.4 Traversing the NLP Processed Words

The NLP processed words returned from the words property are sequenced in the same order as the original text. All punctuation is removed, and except for detected Acronyms, all remaining words are lowercased. The sequenced list of words may be a subset of the original words, depending on the stopwords properties and may be stemmed, lemma, or replaced.

# Get the NLP processed words from the original text as a Python list.  
words = words.words

# Traverse the sequenced list of NLP processed words **for** word **in** words:  
 text = word.word # original or replaced version of the word  
 tag = word.tag # syntactical classification of the word  
 lemma = word.lemma # The lemma version of the word

## 3.5 Stopwords

The properties which determine which words are removed, stemmed, lemmatized, or replaced are set as keyword parameters in the constructor for the Words class. If no keyword parameters are specified, then all stopwords are removed after being stemmed/lemmatized. The list of stopwords is a superset of the Porter list and additionally includes removing additionally syntactical constructs such as numbers, dates, etc. For a complete list, see the reference manual.

If the keyword parameter stopwords is set to False, then all word removal is disabled, while stemming/lemmatization/reducing are still enabled, along with the removal of punctuation. Note in the example below, while stopwords is disabled, the word jumping is replaced with its stem jump.

# No stopword removal  
words = Words(“The lazy brown fox jumped over the fence.”, stopwords=**False**)  
# words => “the”, “lazy”, “brown”, “fox”, “jump”, “over”, “the”, “fence”

# All stopword removal  
words = Words(“The lazy brown fox jumped over the fence.”, stopwords=**True**)  
# words => “lazy”, “brown”, “fox”, “jump”, “fence”

## 3.6 Bare

When the keyword parameter bare is True, all stopword removal, stemming/lemmatization/reducing ad punctuation removal are disabled.

# Bare Mode  
words = Words(“The lazy brown fox jumped over the fence.”, bare=**False**)  
# words => “the”, “lazy”, “brown”, “fox”, “jumped”, “over”, “the”, “fence”, “.”

## 3.7 Numbers

When the keyword parameter number is True, text and numeric version of numbers are preserved; otherwise they are removed. Numbers which are text based (e.g., one) are converted to their numeric representation (e.g., one => 1). The tag value for numbers is set to Vocabulary.NUMBER.

# keep/replace numbers  
words = Words(“one twenty-one 33.7 1/4”, number=**True**)

print(words.words)  
# will output:  
# [  
# { ‘word’: ‘1’, tag: Vocabulary.NUMBER },  
# { ‘word’: ‘21’, tag: Vocabulary.NUMBER },  
# { ‘word’: ’33.7’, tag: tag: Vocabulary.NUMBER },  
# { ‘word’: ‘0.25’, tag: tag: Vocabulary.NUMBER },  
# ]

If a number is followed by a text representation of a multiplier unit (i.e., million), the number and multiplier unit are replaced by the multiplied value.

words = Words(“two million”, number=True)

print(words.words)  
# will output:  
# [  
# { ‘word’: ‘2000000’, tag: Vocabulary.NUMBER},   
# ]

## 3.8 Unit of Measurement

When the keyword parameter unit is True, US Standard and Metric units of measurement are preserved; otherwise they are removed. Both US and EU spelling of metric units are recognized (e.g., meter/metre, liter/litre). The tag value for units of measurement is set to Vocabulary.UNIT.

# keep/replace unit  
words = Words(“10 liters”, number=**True**, unit=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘10’, tag: Vocabulary.NUMBER },   
# { ‘word’: ‘liter’, tag: Vocabulary.UNIT },  
# ]

## 3.9 Standard vs. Metric

When the keyword parameter standard is True, Metric units of measurement are converted to US Standard. When the keyword parameter metric is True, Standard units of measurement are converted to Metric Standard.

# keep/replace unit  
words = Words(“10 liters”, number=**True,** unit=**True** standard=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘2.64172’, tag: Vocabulary.NUMBER },   
# { ‘word’: ‘gallon’, tag: Vocabulary.UNIT },  
# ]

## 3.10 Date

When the keyword parameter date is True, USA and ISO standard date representation and text representation of dates are preserved; otherwise they are removed. Dates are converted to the ISO standard and the tag value is set to Vocabulary.DATE.

# keep/replace dates  
words = Words(“Jan 2, 2017 and 01/02/2017”, date=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘2017-01-02’, tag: Vocabulary.DATE },   
# { ‘word’: ‘2017-01-02’, tag: Vocabulary.DATE },  
# ]

## 

## 3.11 Date of Birth

When the keyword parameter dob is True, date of births are preserved; otherwise they are removed. Date of births are converted to the ISO standard and the tag value is set to Vocabulary.DOB.

# keep/replace dates  
words = Words(“Date of Birth: Jan. 2 2017 DOB: 01-02-2017”, dob=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘2017-01-02’, tag: Vocabulary.DOB },   
# { ‘word’: ‘2017-01-02’, tag: Vocabulary.DOB },  
# ]

If date is set to True without date of birth set to True, date of births will be removed while other dates will be preserved.

## 

## 3.12 Social Security Number

When the keyword parameter ssn is True, USA Social Security numbers are preserved; otherwise they are removed. Social Security numbers are detected from the prefix presence of text sequences indicating a Social Security number will follow, such as SSN, Soc. Sec., Social Security, etc. Social Security numbers are converted to their single 9 digit value and the tag value is set to Vocabulary.SSN.

# keep/replace dates  
words = Words(“SSN: 12-123-1234 Social Security 12 123 1234”, ssn=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘121231234’, tag: Vocabulary.SSN },   
# { ‘word’: ‘121231234’, tag: Vocabulary.SSN },  
# ]

## 3.13 Telephone Number

When the keyword parameter telephone is True, USA/CA telephone numbers are preserved; otherwise they are removed. Telephone numbers are detected from the prefix presence of text sequences indicating a telephone number will follow, such Phone:, Mobile Number, etc. Telephone numbers are converted to their single 10 digit value, inclusive of area code, and the tag value is set to one of:

Vocabulary.TELEPHONE,   
Vocabulary.TELEPHONE\_HOME  
Vocabulary.TELEPHONE\_WORK  
Vocabulary.TELEPHONE\_OFFICE  
Vocabulary.TELEPHONE\_FAX

# keep/replace dates  
words = Words(“Phone: (360) 123-1234, Office Number: 360-123-1234”, telephone=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘3601231234’, tag: Vocabulary.TELEPHONE },   
# { ‘word’: ‘3601231234’, tag: Vocabulary.TELEPHONE\_WORK},  
# ]

## **3.14 Address**

When the keyword parameter address is True, USA/CA street and postal addresses are preserved; otherwise they are removed. Each component in the address is tagged according to the above street/postal address component type, as follows:

* Postal Box (Vocabulary.POB)
* Street Number (Vocabuary.STREET\_NUM)
* Street Direction (Vocabuary.STREET\_DIR)
* Street Name (Vocabuary.STREET\_NAME)
* Street Type (Vocabuary.STREET\_TYPE)
* Secondary Address (Vocabuary.STREET\_ADDR2)
* City (Vocabulary.CITY)
* State (Vocabulary.STATE)
* Postal (Vocabulary.POSTAL)

# keep/replace street addresses  
words = Words(“12 S.E. Main Ave, Seattle, WA”, gender=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘12’, tag: Vocabulary.STREET\_NUM },   
# { ‘word’: ‘southeast’, tag: Vocabulary.STREET\_DIR },   
# { ‘word’: ‘main’, tag: Vocabulary.STREET\_NAME },   
# { ‘word’: ‘avenue’, tag: Vocabulary.STREET\_TYPE },   
# { ‘word’: ‘seattle’, tag: Vocabulary.CITY },   
# { ‘word’: ‘ISO316-2:US-WA’, tag: Vocabulary.STATE },   
# ]

3.15 Gender  
  
When the keyword parameter gender is True, words indicating gender are preserved; otherwise they are removed. Transgender is inclusive in the recognition. The tag value is set to one of Vocabulary.MALE, Vocabulary.FEMALE or Vocabulary.TRANSGENDER .

# keep/replace gender indicating words  
words = Words(“man uncle mother women tg”, gender=**True**)

print(words.words)   
# will output:  
# [  
# { ‘word’: ‘man’, tag: Vocabulary.MALE },   
# { ‘word’: ‘uncle’, tag: Vocabulary.MALE },   
# { ‘word’: ‘mother’, tag: Vocabulary.FEMALE },   
# { ‘word’: ‘women’, tag: Vocabulary.FEMALE },   
# { ‘word’: ‘transgender’, tag: Vocabulary.TRANSGENDER },  
# ]

## 3.16 Sentiment

When the keyword parameter sentiment is True, word and word phrases indicating sentiment are preserved; otherwise they are removed. Sentiment phrases are reduced to the single primary word indicating the sentiment and the tag value is set to either Vocabulary.POSITIVE or Vocabulary.NEGATIVE.

# keep/replace sentiment indicating phrases  
words = Words(“the food was not good”, sentiment=**True**)

print(words.words)   
# will output:   
# [  
# { ‘word’: ‘food’, tag: Vocabulary.UNTAG },  
# { ‘word’: ‘not’, tag: Vocabulary.NEGATIVE},  
# ]

## 3.17 Parts of Speech

When the keyword parameter pos is True, each tokenized word is further annotated with it’s corresponding NLTK parts of speech tag.

# add parts of speech tagging  
words = Words(“Jim Smith”, pos=**True**)

print(words.words)   
# will output:   
# [  
# { ‘word’: ‘food’, ‘tag’: Vocabulary.UNTAG, ‘pos’: NN },  
# { ‘word’: ‘not’, ‘tag’: Vocabulary.NEGATIVE, ‘pos’: NN },  
# ]

## 3.18 Romanization

When the keyword parameter roman is True, the latin-1 character encoding of each tokenized is converted to ASCII.

# Romanization of latin-1 character encodings  
words = Words(“Québec”, roman=**True**)

print(words.words)   
# will output:   
# [  
# { ‘word’: ‘quebec’, ‘tag’: Vocabulary.UNTAG,   
# ]

## 3.19 Bag of Words and Word Frequency Distribution

The property bagsOfWords returns an unordered dictionary of each occurrence of a unique word in the tokenized sequence, where the word is the dictionary key, and the number of occurrences is the corresponding value.

# Get the Bag of Words representation  
words = Words(“Jack and Jill went up the hill to fetch a pail of water. Jack fell down and broke his crown and Jill came tumbling after.”, stopwords=True)  
  
print(words.bagOfWords)  
# will output:  
# { ‘pail’: 1, ‘the’: 1, ‘a’: 1, ‘water’: 1, ‘fetch’: 1, ‘went’: 1, ‘and’: 2, ‘jack’: 2, ‘jill’: 2,  
# ‘down’: 1, ‘come’: 1, ‘fell’: 1, ‘up’: 1, ‘of’: 1, ‘tumble’: 1, ‘to’: 1, ‘hill’: 1, ‘after’: 1 }

The property freqDist returns a sorted list of tuples, in descending order, of word frequencies (i.e., the number of occurrences of the word in the tokenized sequence.

# Get the Word Frequency Distribution  
words = Words(“Jack and Jill went up the hill to fetch a pail of water. Jack fell down and broke his crown and Jill came tumbling after.”, stopwords=True)  
  
print(words.freqDist)  
# will output:  
# [ (‘jack’, 2), (‘jill’, 2), (‘and’, 2), (‘water’, 1), (‘the’, 1), …. ]

# 4. VISION Module

## 4.1 Image Processing

To preprocess an image for computer vision machine learning, you create an Image (class) object, passing as parameters the path to the image, the corresponding label and a path for storing the preprocessed image data, the original image and optionally a thumbnail. The label must be specified as an integer value. Below is a code example.  
  
 **from** vision **import** Image  
 image = Image(“yourimage.jpg”, 101, “storage\_path”)

The above will generate the following output files:  
  
 storage\_path/yourimage.h5 # preprocessed image and raw data and optional thumbnail

The Image class supports processing of JPEG, PNG, TIF and BMP images. Images maybe of any pixel size, and number of channels (i.e. Grayscale, RGB and RGBA).

## 4.2 Image Processing Settings (Config)

CV Preprocessing of the image may be configured for several settings when instantiating an Image object with the optional config parameter, which consists of a list of one or more predefined options.

image = Image(“yourimage.jpg”, 101, “storage\_path”, config=[options])

# options:

gray | grayscale # convert to grayscale (single channel)  
normal | normalize # normalize the pixel data for values between 0 .. 1  
flat | flatten # flatten the pixel data into a 1D vector  
resize=(height,width) # resize the image  
thumb=(height,width) # generate a thumbnail  
nostore # do not store the preprocessed image, raw and   
 thumbnail data

# Example  
image = Image(“image.jpg”, 101, “path”, config=[‘flatten’, ‘thumb=(16,16)’])  
# will preprocess the image.jpg into machine learning ready data as a 1D vector, and  
# store the raw (unprocessed) decompressed data, preprocessed data and 16 x 16

## 4.4 Get Properties of Preprocessed Image Data

After an image has been preprocessed, several properties of the preprocessed image data can be obtained from the Image class properties:

name - The root name of the image.  
 type - The image format (e.g., png).  
 shape - The shape of the preprocessed image data (e.g., (100, 100,3) ).  
 data - The preprocessed image data as a numpy array.  
 raw - The unprocessed decompressed image data as a numpy array.  
 size - The byte size of the original image.  
 thumb – The thumbnail image data as a numpy array.

image = Image(“yourimage.jpg”, “storage\_path”, 101)  
 print(image.shape)  
 # Will output something like:  
 # (100,100,3)

## 4.5 Asynchronous Processing

To enhance concurrent execution between a main thread and worker activities, the Image class supports asynchronous processing of the image. Asynchronous processing will occur if the optional parameter ehandler is set when instantiating the Image object. Upon completion of the processing, the ehandler is called, where the Image object is passed as a parameter.

**def** done(i):  
 “”” Event Handler for when processing of image is completed “””  
 print(“DONE”, i.image)

# Process the image asynchronously  
image = Image(“yourimage.png”, “storage\_path”, 101, ehandler=done)

## 4.6 Image Reloading

## Once an Image object has been stored, it can later be retrieved from storage, reconstructing the Image object. An Image object is first instantiated, and then the load() method is called specifying the image name and corresponding storage path. The image name and storage path are used to identify and locate the corresponding stored image data.

# Instantiate an Image object  
image = Image()

# Reload the image’s data from storage  
image.load( “myimage.png”, “mystorage” )

## 4.7 Image Collection Processing

To preprocess a collection of images for computer vision machine learning, you create an Images (class) object, passing as parameters a list of the paths to the images, a list of the corresponding label and a path for storing the collection of preprocessed image data, the original images and optionally thumbnails. Each label must be specified as an integer value. Below is a code example.  
  
 **from** images **import** Images  
 images = Images([“image1.jpg”, “image2.jpg”], “train”, labels=[101, 102], name=’ c1’)

The above will generate the following output files:   
  
 train/c1.h5 # preprocessed image data  
  
The Images object will implicitly add the ‘nostore’ setting to the configuration parameter of each Image object created. This will direct each of the Image objects to not store the corresponding image data in an HD5 file.   
  
Instead, upon completion of the preprocessing of the collection of image data, the entire collectio of preprocessed data is stored in a single HD5 file.

## 4.8 Image Collection Processing Settings (Config)

Configuration settings supported by the Image class may be specified as an optional parameter to the Images object, which are then passed down to each Image object generated for the collection.   
  
 # Preprocess each image by normalizing the pixel data and then flatten into a 1D vector  
 images = Images([“image1.jpg”, “image2.jpg”], “train”, labels=[101, 102], config=[‘normal’, ‘flatten’])

## 4.9 Get Properties of a Collection

After a collection of images has been preprocessed, several properties of the preprocessed image data can be obtained from the Images class properties:

name – The name of the collection file.  
 time – The time to preprocess the image.  
 data – List of Image objects in the collection.  
 len() – The len() operator will return the number of images in the collection.  
 [] – The index operator will access the image objects in sequential order.

# Access each Image object in the collection  
**for** ix **in** range(len(images)):  
 image = images[ix]

## 4.10 Splitting a Collection into Training and Test Data

Batch, mini-batch and stochastic feed modes are supported. The percentage of data that is training (vs. test) is set by the split property, where the default is 0.8. Optionally, a mini-batch size is set by the minibatch property. Prior to the split, the data is randomized.

The split() property when called as a getter will return the training data, training labels, test data, and test labels.

# Set 70% of the images in the collection to be training data  
images.split = 0.7  
# Get the entire training and test data and corresponding labels as lists.  
X\_train, X\_test, Y\_train, Y\_test = images.split

Alternately, the next() operator will iterate through the image data, and corresponding label, in the training set.

# Set 70% of the images in the collection to be training data  
images.split = 0.7  
# Iterate through the training data **while** ( data, label = next(images) ) **is not None**:  
 **pass**

Training data can also be fetched in minibatches. The mini batch size is set using the minibatch property. The minibatch property when called as a getter will return a generator. The generator will iterate through each image, and corresponding label, of the generated mini-batch. Successive calls to the minibatch property will iterate through the training data.

# Set 70% of the images in the collection to be training data  
images.split = 0.7  
# Train the model in mini-batches of 30 images  
images.minibatch = 30  
# loop for each mini-batch in training data  
**for** \_ **in** range(nbatches)  
 # create the generator  
 g = images.minibatch  
 # iterate through the mini-batch  
 **for** data, label in **g**:  
 **pass**

## 4.11 Asynchronous Collection Processing

To enhance concurrent execution between a main thread and worker activities, the Images class supports asynchronous processing of the collection of images. Asynchronous processing will occur if the optional parameter ehandler is set when instantiating the Images object. Upon completion of the processing, the ehandler is called, where the Images object is passed as a parameter.

**def** done(i):  
 “”” Event Handler for when processing of collection of images is completed “””  
 print(“DONE”, i.images)

# Process the collection of images asynchronously  
images = Images([“img1.png”, “img2.png”], “train”, labels=[0,1], ehandler=done)

## 4.12 Collection Reloading

## Once an Images object has been stored, it can later be retrieved from storage, reconstructing the Images object, and corresponding list of Image objects. An Images object is first instantiated, and then the load() method is called specifying the collection name and corresponding storage path. The collection name and storage path are used to identify and locate the corresponding stored image data.

# Instantiate an Images object  
images = Images()

# Reload the collection of image data from storage  
images.load( “mycollection”, “mystorage” )

# 4. SEGMENTATION Module

The segmentation module is newly introduced in Gap v0.9 prelaunch. It is in the early stage, and should be considered experimental, and not for commercial-product-ready yet. The segmentation module analyzes the whitespace layout of the text to identify the ‘human’ perceived grouping/purpose of text, such as paragraphs, headings, columns, page numbering, letterhead, etc., and the associated context.

In this mode, the text is separated into segments, corresponding to identified layout, where each segment is then NLP preprocessed. The resulting NLP output is then hierarchical, where at the top level is the segment identification, and it’s child is the NLP preprocessed text.

## 4.2 Text Segmentation

When the config option ‘segment’ is specified on a Document object, the corresponding text per page is segmented.

# import the segmentation module  
**from** segment import Segment

segment = Segment(“para 1\n\npara 2”)  
print(segment.segments)  
# will output:  
# [ { ‘tag’: 1002, words: [ { ‘word’: ‘para’, ‘tag’: 0}, {‘word’: 1, ‘tag’: 1}]},  
# { ‘tag’: 1002, words: [ { ‘word’: ‘para’, ‘tag’: 0}, {‘word’: 2, ‘tag’: 1}]}  
# ]

**Proprietary and Confidential Information  
Copyright ©2018, Epipog, All Rights Reserved**