# Philosophy

HistoQC consists of a pipeline of modules sequentially applied to an image. These modules act on the image to (a) produce metrics, and (b) produce output images after applying thresholds or running classifiers.

When an image is loaded it is initially assigned a “True mask” indicating that every pixel in the image is artifact free and “useful” for analysis. This mask is referred to as *img\_mask\_use*.

The HistoQC approach uses the specified pipeline to sequentially refine the *img\_mask\_use* mask. For example, while initially the entire image is considered useful, after the “ LightDarkModule.getIntensityThresholdPercent:tissue” module is run, the background of the tissue should now be set to false, refining the locations in the image which are suitable for computation and analysis.

As such, the order of the events in the pipeline is important as the regions considered for computation may be affected. In particular, most modules have the option to “limit\_to\_mask” which implies that the module’s operations will only take place in the regions currently identified as accepted by *img\_mask\_use*. For example, when computing image color distributions, one would like to only operate on the part of the image that has tissue and avoid the white background which will artificially inflate the white value of the distribution, thus placing “HistogramModule.getHistogram” after “LightDarkModule.getIntensityThresholdPercent:tissue” is ideal.

# Suggested workflow

Through various experiments, we have come to the following suggested workflow. Depending on your task and the expected homogeneity of your dataset, this approach may be rather extreme, so it is suggested that you modify your approach accordingly.

1. Run HistoQC on all images using a minimal pipeline, such as the one contained in config\_first.ini. This allows for discovery of images which are scanned at different magnifications, e.g., 20x and 40x images. Additionally it performs basic tissue detection (white threshold) and subsequently computes histograms and differences to target templates. Using this information, split the cohort into sub-cohorts based on these values since (a) various modules are likely to function differently at different magnifications, (b) the *image level* which is loaded by openslide will be different, implying potential memory issues (loading too big of an image accidently) or attempting to open a level which doesn’t exist.

Ideally, one wants to create sub-cohorts which have the same magnifications, contain the same number of internal storage format levels (“levels”), and share similar appearance properties (lightness, stain intensity, etc)

This can be done easily using the web user interface and the parallel coordinates graph. Clicking and dragging on any of the axis allows for the creation of filters which update both the table above and the images below. Dragging an existing filter up and down dynamically adjusts the filters. When looking at a filtered view of the table one can click “Save Filtered” and save just that particular subset of images.

Additionally, if there are images which should be removed, multi-selecting them and clicking “Delete” will remove them from the table. Subsequently clicking save will result in a subset of the output file.

2. Once the sub-cohorts are built, you can rerun the pipeline using an expanded set of models which have higher computational load. An example of the full pipeline is in config\_40x\_HE\_full.ini, designed to work with H&E images at 40x. Various configuration options are discussed below. Here again, we can identify images which are either not suitable for computation due to artifacts but also we can determine if the suggested masks are appropriate for the desired downstream computational tasks. An easy way of doing this is to click on the “compare” drop down and select “\_fuse.png”, which will show the original image next to the fused images.

In case of errors: For example, if some images caused errors because of out of memory, you can rerun pipeline simply by deleting their output directories. they are easily found because they don't have thumbnail images (which are created in the last step in all pipelines). matlab code to do this:

files=dir('\*.svs');

for zz=1:length(files)

fname=files(zz).name;

if(~exist(sprintf('%s/%s\_thumb.png',fname,fname),'file'))

fprintf('%s\n',fname);

rmdir(fname,'s');

end

end

# Pipeline configuration

## Pipeline module order

The pipeline configuration is specified using a configuration file. A default config.ini is supplied in the repository. The configuration syntax is that of pythons configparser [https://docs.python.org/3/library/configparser.html]. In brief this means that the configuration file has sections, and each section has key value parameters. In the HistoQC setting, the sections are named for their associated module.

There is only a single required section, which is called “[pipeline]”. This section defines, again in sequential order, the steps which will be taken on a per image basis. An example pipeline configuration is presented here:

**[pipeline]**steps= BasicModule.getBasicStats  
 BasicModule.getMag  
 ClassificationModule.byExampleWithFeatures:pen\_markings  
 #ClassificationModule.byExampleWithFeatures:pen\_markings\_red  
 ClassificationModule.byExampleWithFeatures:coverslip\_edge  
 #LightDarkModule.getIntensityThresholdPercent:bubble  
 LightDarkModule.getIntensityThresholdPercent:tissue  
 #BubbleRegionByRegion.pixelWise  
 LightDarkModule.getIntensityThresholdPercent:darktissue  
 MorphologyModule.removeSmallObjects  
 MorphologyModule.fillSmallHoles  
 BlurDetectionModule.identifyBlurryRegions  
 BasicModule.finalProcessingSpur  
 BasicModule.finalProcessingArea  
 HistogramModule.compareToTemplates  
 HistogramModule.getHistogram  
 BrightContrastModule.getContrast  
 BrightContrastModule.getBrightness  
 DeconvolutionModule.seperateStains  
 SaveModule.saveFinalMask  
 SaveModule.saveThumbnail  
 BasicModule.finalComputations

We note here that it is possible to use the *same* module multiple times, with different settings and assign it a different name. For example getIntensityThresholdPercent applies a threshold to the image, and “getIntensityThresholdPercent:tissue” applies a high threshold to remove the background on the slide, while “getIntensityThresholdPercent:darktissue” applies a low threshold to identify regions which may contain artifacts such as folded tissue or drastic overstaining. Each instance of the module is defined as the base module name (getIntensityThresholdPercent) plus a double colon followed by the specific instance name of that module (e.g., “:darktissue”). Later on in the configuration file, we can see the associated sections are named exactly the same ([LightDarkModule.getIntensityThresholdPercent:darktissue]), and that each section contains a “name:” parameter, which is used as the output name of the image as well as the column name in the tsv results file.

## Pipeline image size

The BaseImage section’s image\_work\_size parameter specifies the default size of the internal representation of the image to be used in the pipeline. Most modules, unless otherwise specified, will use an image of this size to perform their operation, thus setting a suitable size is important. In most cases, it is infeasible to load an entire 40x whole slide mount, but even having done so would not provide greater specificity in many of the metrics (e.g., color distributions). As such a default of “3” is recommend, which when using Aperio svs files typically coincides with a base image magnification of 40x being loaded at 1.25x (1:32 reduction).

There are 4 ways to specify the desired image size

When image\_work\_size < 1 and is a floating point number, it is considered a downscaling factor of the original image (e.g., new.image.dimensions = image.dimensions \* image\_work\_size)

if image\_work\_size < 100, it is considered to indicate the *level* of image to load using the openslide pointer. In the case of Aperio Svs, this typically coincides with {0=Base, 1 = 4:1, 2=16:1, 3=32:1, etc}

if image\_work\_size > 100, this is considered to be the *exact* longest dimension desired (e.g., an image of size 1234 x 2344, if image\_work\_size is set to 500, the output will be 263 x 500). Note this will cause different magnifications per image (if they’re of different sizes)

If image\_work\_size = 1.25x, this is considered as the desired apparent magnification. On one hand, this makes processing a bit easier, as each image, regardless of its base magnification, will be made to have the same apparent magnification but this comes with 2 caveats: (1) the computation time to generate each of these images could be 1 minute or mode as the next higher level magnification needs to be loaded and literately down sampled to the desired magnification (in cases of going from 2x to 1.25x this is rather trivial but going from 5x to 1.25x can take a few moments), (2) one should really consider if their downstream analytics are capable of handling heterogeneity (otherwise its best to split images by base magnification and base number of levels). This approach is different than #1, as #1 directly loads the next highest magnification and then resizes it downwards, potentially exploding memory, this approach sequentially loads smaller tiles, resizes them, and then merges them together, drastically reducing memory overhead.

BEWARE: these operations are not free! In cases #1 and #3, we leverage the openslide “get\_thumbnail” function to produce the requested image. This function works by taking the *next* largest image layer, loading it, and then downsizing it to the requested size. One can image that if the image\_work\_size size is not properly set, the whole uncompressed image will be loaded before down sampling and thus likely exploding available resources.

# Adding classification type modules

Most of the modules are implemented using statistics or thresholds and are thus relatively easy to setup. The classification modules represent a departure from that simplicity and are not only the most sophisticated modules in HistoQC, but also the most powerful. The classification approach consists of first loading exemplar images from which to create a model. Each exemplar should consists of 2 images of the same size, the first the original image and the second a binary mask. Each set is specified under the “examples” parameter, one per line, and each separated by a double colon like so:

examples: ./pen/1k\_version/pen\_green.png:./pen/1k\_version/pen\_green\_mask.png  
 #./pen/1k\_version/pen\_red.png:./pen/1k\_version/pen\_red\_mask.png

Which indicates that relative or absolute location of 2 exemplars (pen\_green and pen\_red), and their associated mask (pen\_green\_mask.png and pen\_red\_mask.png). The mask is a binary image (i.e., only containing the values 0,1) identifies which pixels should be used as the positive class in the image (e.g., 1), and the pixels which should be used as the negative class (e.g., 0). It usually makes sense for these images to be of the same magnification specified by “image\_work\_size”, as this will improve the performance of the classifier.

In the second step, after the images are loaded, a classifier is trained. To improve the robustness of the classifier, we allow for the computation of a number of different pixel-features to augment the original RGB space. These features are those implemented in skimage.filters [http://scikit-image.org/docs/dev/api/skimage.filters.html] and include:

features: frangi  
 laplace  
 rgb  
 #lbp  
 #gabor  
 #median  
 #gaussian

Each of their parameters can be set by using the feature name as the prefix to the parameter, for example: “frangi\_black\_ridges: True”, sets the “black\_ridges” parameter of the frangi filter to true. A single model is trained and shared by all individual the processes which request access to it reducing memory and training efforts.

After the model is trained, it is retained in memory, and is applied at the appropriate time to the images identified by HistoQC. Internally, the output from this is a probability likelihood that a particular pixel belongs to the trained positive class, but as a real value output is not suitable here, we accept a parameter “tresh” which will apply a threshold to the probability map to provide the final binary value mask which is used in downstream analysis.

# Remotely Viewing

If the analysis and data is performed on a remote server, it is possible to view the results in the UI without downloading a copy of all of the output images locally. This is slightly more complex, but not burdensome

1. Download a local copy of the results.tsv file

2. Make sure the HistoQC/currentVersion/Data directory contains a link (or copy) of the directory created by the qc\_pipeline.py

3. Launch a python simple HTTP server (this is a build-in python module), and note the port it is listening on:

HistoQC/currentVersion$ python -m SimpleHTTPServer!!!

Serving HTTP on 0.0.0.0 port 8000 ...

4. Go to <http://ip_address:8000>, you should be able to see the HistoQC user interface, from there select the result file you downloaded locally.

5. Interact with user interface in the normal fashion

# Extending HistoQC

HistoQC was specifically designed to be very modular and allow for easy extensibility by even novice programmers. For new functionality, it is recommended to look at the available modules, identify one which is most similar in functionality to the new target functionality, and use that as a basis for future development.

Here we will describe the components necessary to have a fully functioning module.

## Naming

The filename of the new module should be descriptive of the class of the module. For example, “HistogramModule” consists of functionality associated with histograms. The filename is thus HistogramModule.py. Inside of this file, we can define individual functions, for example “compareToTemplates”, which loads templates and compares their distributions to the image’s distributions. To add the module to the pipeline, we simply need to add a line of the format *filename*.*function* to the [pipeline] section in the “steps” list, and at run time, HistoQC will dynamically load this function into the memory space. In this example, we would add “HistogramModule.compareTemplates” to the list. Note that the “py” extension has been removed from the filename. By adding a section named [HistogramModule.compareTemplates] to the bottom of the configuration file, we can supply parameters which will automatically become available at function execution time.

## Internal Representation (default variables)

Looking at the HistogramModule.compareTemplates function mentioned above we can see the function’s prototype has two parameters:

*def compareToTemplates(s, params):*

*All* modular functions in the pipeline list receive these two parameters (i.e., s and params), and thus are the keys to communicating and storing information in HistoQC.

### Params

Params contains the parameters for that specific modular function as specified in the configuration file. Any values added to here will be lost after the function exists. They can be accessed in the function as a standard python dictionary, but using the *get* method with a reasonable default and appropriate casting is *highly* suggested:

*thresh = float(params.get("threshold", .5))*

In this example, we load the variable “threshold” from the file. If it doesn’t exist, we assume a default value of .5. In both cases we case the result to a float, as the configuration parser may potentially return a string as opposed to the desired type.

### S

“s” is a hold-all dictionary with 1 instance per image and is of type BaseImage. It contains all of the metrics, metadata, and masks. Most importantly it contains an already opened openslide pointer for usage in loading the slide. There are some default keys and functions provided for your usage which cover most operations. The default keys are discussed here:

|  |  |
| --- | --- |
| Key | Description |
| s["warnings"] | Append any warnings to this field and they will appear in the tsv file under the “warnings” column. Used for informing the user that things in a particular module may not have gone as expected |
| s["filename"] | The filename of the image |
| s["outdir"] | The location of the directory for the particular image, useful for saving masks |
| s["os\_handle"] | The pre-opened openslide handle. It is possible to use this directly, but for more robust access, one might consider using the *getImgThumb* described below |
| s["image\_work\_size"] | Discussed in the above section, specifies the default image working size |
| s["img\_mask\_use"] | A binary mask indicating where at this stage in the pipeline HistoQC believes the artifact tissue to be |
| s["comments"] | This is typically left blank so that the front end or downstream user has a dedicated column for their comments already available in the spreadsheet, but regardless may be added to if additional information is warranted |
| s["completed"] | This keeps an automatically updated list of the modules which have been completed (by name), allowing for the enforcement of prerequisites |

The available functions are discussed here:

|  |  |
| --- | --- |
| Function | Description |
| addToPrintList(name, val) | Providing the name and the value (in string format) will dynamically add this value to the output tsv file, and will also appear in the front end |
| getImgThumb(dim) | As discussed above in Pipeline image size, this will obtain a numpy representation of the underlying image. The additional functionality here also will cache the image locally so that subsequent requests for the image at that size will return immediately as opposed to requiring additional computation time to produce. |

## Saving output images

Examining output images is one of the most important features of HistoQC, as a result adding additional output images is easily done. Here we examine a line of code which saves an output image (where *io* is imported from skimage):

*io.imsave(s["outdir"] + os.sep + s["filename"] + "\_BubbleBounds.png", mask.astype(np.uint8) \* 255)*

First, we can see that the location and filename consist of:

*s["outdir"] + os.sep + s["filename"]*

This should never be changed unless there is a strong reason to do so. Next we add an underscore followed by the name of the particular mask we’re producing, in this case “\_BubbleBounds.png”. Afterwards, we provide the matrix to be saved, in this case a binary mask (0s and 1s) which is converted to 0 and 255 for easier downstream usage.

To have the new image type appear in the front-end user interface, the suffix needs to be manually added. Open global\_config.js, scroll to the definition of DEFAULT\_IMAGE\_EXTENSIONS (around line 20), and add the new suffix to the list. That’s it!