1047904_Data Analytics at Scale

January 12, 2021

1 Submission for Data Analytics at Scale

1.0.1 Candidate number: {1047904}

Word Count (excluding Tables, Appendices, References): **3463**

[9]:
```
%%html
<!--Make following tables left-aligned-->
<style>
table {float:left}
</style>
```

```
<IPython.core.display.HTML object>
```

2 Executive Summary

FIN commissioned *Byte Right* to conduct a two-staged performance assessment of its in-house image hashing algorithm `FINd`. For the first part, `FINd` was profiled and three candidate optimizations (`numpy`, `numpy_numba`, `numpy_scipy`) were derived to remedy two major function bottlenecks. In hindsight, two optimizations grounded on *Numpy* in combination with *Numba* and *Scipy* respectively led to dramatic efficiency gains while a *Numpy*-only optimization failed to gain traction. For the second part, `FINd` optimization variant `numpy_numba` was compared against `ahash` and `dhash`, two alternative hashing approaches from the *imagehash* library. Performance appraisals qualified `ahash` and `dhash` as computationally superior while accuracy appraisals hint at `np_numba`'s potential to yield highly competitive results provided the right hamming distance threshold is set. Faceted findings motivate the recommendation to combine multiple approaches in favour of more reliable classifications.

3 Part I

Part 1 commences with a brief recap of the algorithm in scope. Next, the underlying assessment framework guiding conducted performance and accuracy appraisals is outlined. The results section opens with an exposition of computational shortcomings identified within `FINd` before justifying three candidate optimizations pursued to counter unearthed bottlenecks. Competing implementations are then compared for accuracy and performance based on the adopted framework.

3.1 Understanding FINd

The baseline script was translated into a technical architecture diagram providing a bird's eye view of the algorithm's main functions, their scope of duty and hypothesized relevance for the profiling exercise (*Figure 1*).

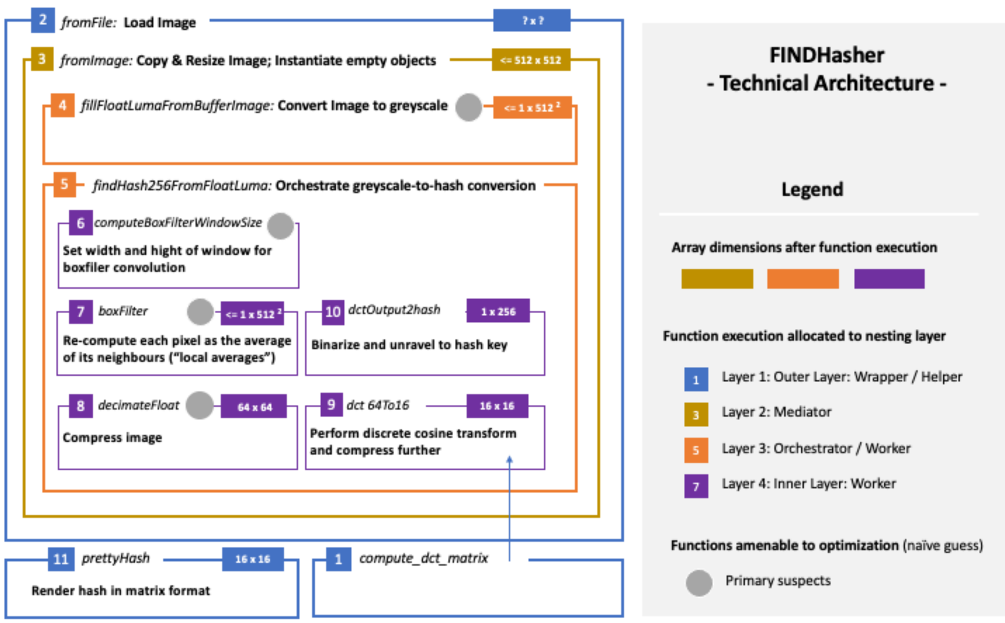


Figure 1: *Technical Architecture Diagram FINd*

The hash grounds on 11 functions distributing across 4 layers. While functions `fromFile`*(2)*, `fromImage`*(3)* `findHash256FromFloatLuma`*(5)* and `prettyHash`*(11)* mainly emerge as wrapping / orchestrating functions, the analytical workbench largely distributes across functions `fillFloatLumaFromBufferImage`*(4)* and a collection of five sub-routines *(6-10)* coordinated by `findHash256FromFloatLuma`*(5)* . A preliminary briefing pointed out the relative maturity of functions `dct64To16` *(9)*, `compute_dct_matrix` *(1)* and `dctOutput2hash` *(10)*. Consequently, four functions emerged as primary suspects for optimization.

3.2 Evaluation Framework Introduction

A 5-step assessment framework was conceived to evaluate FINd and optimized variants. Framework components are outlined below:

3.2.1 Image Sampling

A disproportionate stratified sampling routine was slotted in ahead of any comparative assessments. Sampling at random is meant to prevent any unobserved factors (e.g. image contents, colour schemes) from subliminally conditioning devised testing outcomes. Averaging profiling runs over a sample of multiple images was further expected to yield more robust outcomes to reason on. Disproportionate stratified sampling was favoured over simple random sampling primarily to empower equal-sized within- and across strata hash comparisons (Daniel, 2012).

Eventually, a random subset of 3 image groups from 1035 available strata was drawn. Per image group, 5 images were sampled at random. Random seeds were employed to guarantee results reproducibility for FIN after project handover.

3.2.2 Unit Testing

To flag premature optimizations, a unit test calculating pairwise hamming distances between hash representations of FINd and any optimization variant was performed across all sample images. In result, any "false positive" optimizations which are computationally efficient but generate inaccurate or erroneous hashes could be screened out right from the start.

3.2.3 Performance Measurement

First, program execution times were captured for each version of `FINd` and averaged across sample images using `timeit`. Overall runtime results further served as a ballpark figure to rank competing versions for computational efficiency.

Second, each candidate version was screened for function-level computational bottlenecks using `cProfile` to eye areas for optimization. Bottleneck qualification grounds on inspections of output metric `tottime` which quantifies the fraction of runtime spent in a particular function excluding sub-function calls.

Third, `line_profiler` was used to spot within-function bottlenecks and inspire concrete changes to the code baseline. In this context, the out-of-the-box usage of the `@profile` decorator appeared to confound `cProfile` measurements and exert material impact on test case runtimes. Direct recourse to the `LineProfiler` API was found to alleviate observed complications and sustain the feasibility of the assessment framework when consecutively profiling across all sample images.

3.3 Discussion of Results

The following section discusses the profiling results on `FINd` (thereafter referred to as `FINd_base`), sketches out attempted optimizations and contrasts results in a comparative analysis.

3.3.1 Profiling Results on FINd

Timing `FINd_base` for a stratified sample of 15 images yields an average execution time of 458.6 ms *(Figure 3)*. An examination of the `cProfile` output *(Figure 4, Subplot 1)* reveals that the two functions `boxFilter` and `fillFloatLumaFromBufferImage` jointly account for around 95% of `FINd_base`'s overall execution time.

Line profiling shed light on the taxing statements within these function calls. Within `fillFloatLumaFromBufferImage` *(Table 1)*, a non-trivial amount of time 9.4% (`Line #72`) is allotted to executing the inner for-loop. More shockingly, almost 68.9% of time is spent on the piecemeal retrieval of the RGB values per pixel (`Line #73`) while another 21.5% of time is apportioned to the element-wise transformation to grayscale and subsequent index-based allocation within a designated list (`Line #74`).

Line #	Hits	Time	Per Hit	% Time	Line Contents
67					`def fillFloatLumaFromBufferImage(self, img, luma):`
68	1	2.0	2.0	0.0	`numCols, numRows = img.size`
69	1	143.0	143.0	0.1	`rgb_image = img.convert("RGB")`
70	1	1.0	1.0	0.0	`numCols, numRows = img.size`
71	251	96.0	0.4	0.0	`for i in range(numRows):`
72	62750	22475.0	0.4	9.4	`for j in range(numCols):`
73	62500	164006.0	2.6	68.9	`r, g, b = rgb_image.getpixel((j, i))`
74	62500	51175.0	0.8	21.5	`luma[i * numCols + j] = ( self.LUMA_`

Table 1: *Line Profiler Results: fillFloatLumaFromBufferImage - FINd_base*

A similar pattern emerges from the line profiling results for `boxFilter` *(Table 2)*, where the computational burden arising from a total of four interlaced for-loops skews the runtime distribution. Concretely, more than 45% of function time is due to iterating over length and width of the moving window (`Line #177 & #178`) while just as much time is spent on the stepwise summation of each pixel within a given window (`Line #179`).

Line #	Hits	Time	Per Hit	% Time	Line Contents
165					`@classmethod`
166					`#@profile`
167					`def boxFilter(cls,input,output,rows,cols,rowWin,colWin):`
168	1	2.0	2.0	0.0	`halfColWin = int((colWin + 2) / 2)  # 7->4, 8->5`
169	1	1.0	1.0	0.0	`halfRowWin = int((rowWin + 2) / 2)`
170	251	109.0	0.4	0.0	`for i in range(0,rows):`
171	62750	20754.0	0.3	0.8	`for j in range(0,cols):`
172	62500	21223.0	0.3	0.8	`s=0`
173	62500	35435.0	0.6	1.4	`xmin=max(0,i-halfRowWin)`
174	62500	33499.0	0.5	1.3	`xmax=min(rows,i+halfRowWin)`
175	62500	33190.0	0.5	1.3	`ymin=max(0,j-halfColWin)`
176	62500	32764.0	0.5	1.3	`ymax=min(cols,j+halfColWin)`
177	435250	162863.0	0.4	6.5	`for k in range(xmin,xmax):`
178	2595831	991159.0	0.4	39.3	`for l in range(ymin,ymax):`
179	2223081	1149471.0	0.5	45.6	`s+=input[k*rows+l]`
180	62500	41067.0	0.7	1.6	`output[i*rows+j]=s/((xmax-xmin)*(ymax-ymin))`

Table 2: *Line Profiler Results: boxFilter - Find_base*

These findings epitomise major shortcomings of the interpreted, dynamic nature of pure python implementations which pile up significant overhead in functions primarily occupied with the recurring execution of trivial operations. More precisely, recorded overhead accrues from repeat object type evaluations and function dispatches at each iteration rather than being a product of arithmetic complexity (Vanderplas, 2016). While both functions suffer the same shortcoming, issues compound for `boxFilter` where each pixel is called multiple times, hence stretching inefficiencies over almost 2.6 million hits for the 3rd nested for-loop (`Line# 178`).

3.3.2 Candidate Optimizations

A total of three optimizations were brought to trial to address identified bottlenecks.

Optimization with *Numpy*

Since `FINd_base` rests on the recurring application of trivial operations across homogenous data structures, a first optimization was attempted using *Numpy*. Written in C, *Numpy* is equipped with compiled, statically typed routines which forgo repeat type-checking at runtime and alleviate computational burden of nested for-loop structures through broadcasting. The code was hence modified in that instantiated objects were type-casted to arrays and loops were (partially) replaced with vectorized implementations grounded on *Numpy* universal functions (ufunc).

Optimization with *Numba*

To capitalize on previously included *Numpy* ufuncs, a second optimization was attempted using *Numba*. Concretely, *Numba* compiles targeted functions to intermediate representation at runtime using the low-level virtual machine (*LLVM*) tool set (Lanaro, 2000). *Numba* was not solely chosen for likely synergies with *Numpy* but also for its tolerance towards code sections not amenable to native code conversions (Lanaro, 2000) as well as for its ability to overcome *Numpy*'s constraint to single threaded executions by means of parallelizing array expressions (Numba.pydata.org, 2020a). To achieve the aspired speed-ups, `boxFilter` and `fillFloatLumaFromBufferImage` were decorated accordingly.

Optimization with *Scipy*

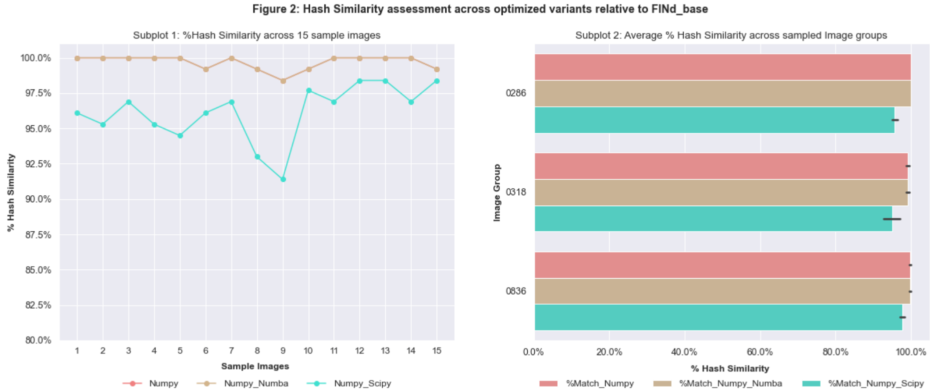
Since function re-engineering using *Numpy* retained three of the four original loops, a third and final optimization was attempted using *Scipy*. The idea was to exhaustively substitute taxing loop operations with computational routines written in C and Fortan rendered accessible via *Scipy* function wrappers (Scipy.org, 2020a). Specifically, *Scipy* 's signal processing module `convolve2d` was used, representing a bespoke convenience function to convolve two-dimensional arrays (Scipy.org ,2020b)

The report proceeds with discussing profiling outputs for the aforementioned optimization attempts.

3.3.3 Comparative Analysis of Candidate Optimizations

Unit test results confirm that performed optimizations did not entail substantial losses in accuracy *(Figure 2)*.

Figure 2: Hash Similarity assessment across optimized variants relative to FINd_base

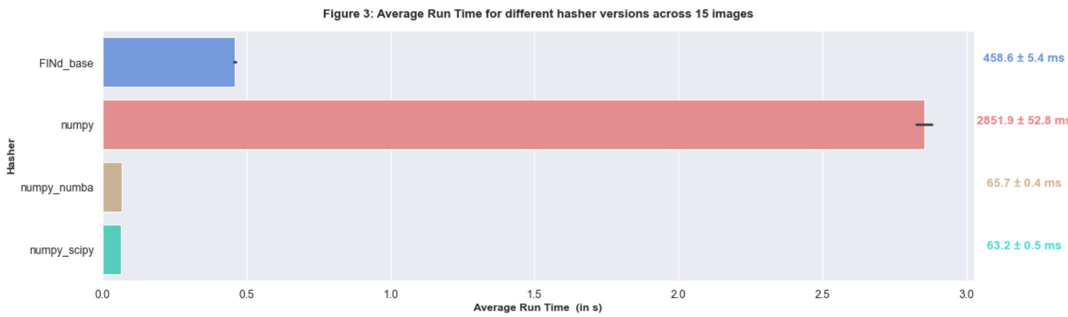


Relative hash similarities to `FINd_base` $>98\%$ were found across all sample images for hash versions `numpy` and `numpy_numba` which perfectly overlay at all instances *(Figure 2, Subplot 1)*. `numpy_scipy` shows similarity scores in the higher 90%'s except for three hash representations marked by 5% to 9% deviations to `FINd_base` *(Figure 2, Subplot 1)*. Observed discrepancies were suspected to result from divergent rounding conventions in *Numpy* and *Scipy* (based on *Numpy*) vs. pure Python which were carried forward from modified functions to the output.

Figure 2 further reveals that hash outcomes from `numpy` and `numpy_numba` are largely image-group agnostic whereas hash key congruencies to `FINd_base` seem to vary between images as indicated by different average similarities and standard error bars for `numpy_scipy` across the three sampled image groups *(Subplot 2)*.

Eventually, unit test results were deemed satisfactory to invite all candidates to subsequent execution time assessments.

An investigation of averaged program execution runtimes reveals sevenfold decreases in runtime for hasher versions `numpy_numba` and `numpy_scipy` relative to `FINd_base` *(Figure 3)*.



`cProfile` outputs confirm that both optimization attempts succeeded in alleviating the computational overhead diagnosed in `boxFilter` and `fillFloatLumaFromBufferImage` either largely (for `numpy_numba`) or entirely (for `numpy_scipy`) *(Figure 4, Subplot 3 and 4)*.

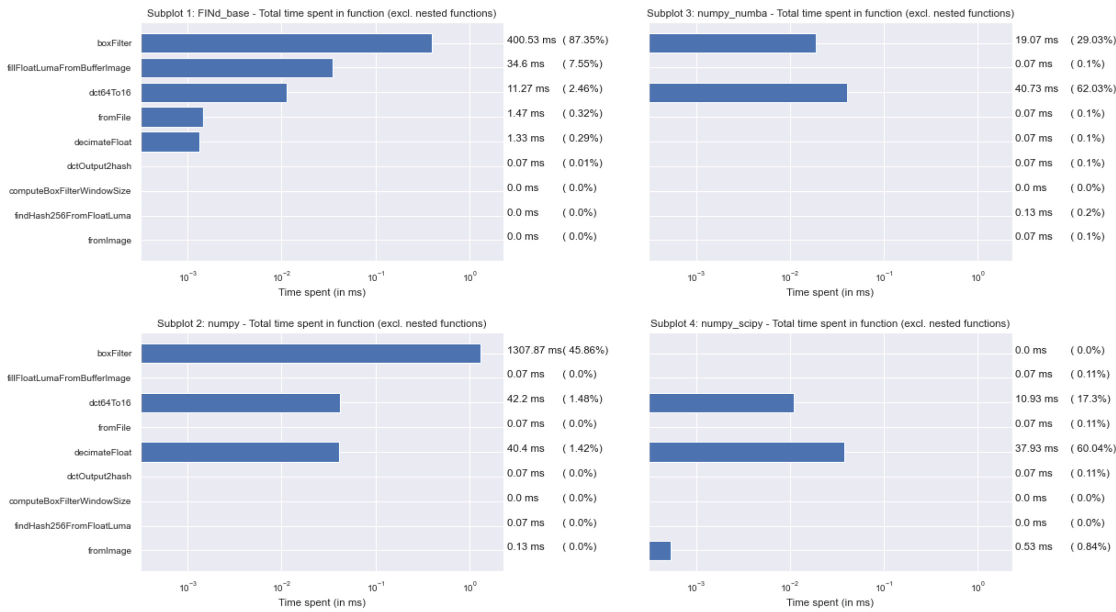
`Line_profiler` outcomes for `numpy_scipy` further reinforce this conclusion by showing that both functions are henceforth exempt from any overt loop-like operations deprived of computational purpose *(Appendix 1, Table 1)*.

Noteworthily, `line_profiler` failed to output actionable results for `numpy_numba`. This demonstrates a well-known compatibility issue between compiled code and pure Python tooling (stackoverflow, 2018) and highlights a trade-off between optimization and post-hoc code investigability when working with high-performance compilers like *Numba*.

In regards of anticipated parallelization gains using *Numba*, a marginal improvement was achieved on `fillFloatLumaFromBufferImage`. Yet, parallelization endeavours were eventually forgone for `boxFilter`. Contrary to the assumption that the presence of parallel semantics are unequivocally amenable to concurrent execution (Numba.pydata.org, 2020b), a substantial deterioration in runtime from ~65 ms up to ~2.5 s was observed when the parallelization argument was passed to the *Numba*'s `@jit` decorator. Since parallelization was attempted for relatively small-sized arrays and trivial arithmetic, resulting speedups might eventually fail to compensate for overhead incurred for thread dispatch (stackoverflow, 2017).

Lastly, warnings were issued during initial *Numba* compile, disclosing the continued existence of unsupported function elements forcing *Numba* out of native mode back into object mode (Lanaro, 2000). While this highlights additional room for improvement, the magnitude of efficiency gains achieved relative to `FINd_base` again emphasizes the flexibility of Numba to effectively mediate between both worlds *(Figure 3)*.

Figure 4: Total Time spent across key functions for different versions of FINd (15 images)



To our surprise, the *Numpy*-only optimization turned out to lose the race by a large margin. With an average program execution time of ~2.85 s *(Figure 3)* , hasher version *numpy* was not only more than six times slower than `FINd_base` but lacked orders of magnitude behind competing optimizations `numpy_numba` and `numpy_scipy`.

While applied vectorization proved successful in alleviating for-loop induced overhead within `fillFloatLumaFromBufferImage`, optimization attempts backfired on `boxFilter`. With a runtime of ~1.3 s, "numpyrized" `boxFilter` was observed to run almost three times as long as the original algorithm `FINd_base` altogether *(Figure 4, Subplot 2)*. Associated `line_profiler` output further reveals that time per hit substantially increased for all function elements of `boxFilter` relative to `FINd_base`*(Appendix 1,Table 2)*. This skyrockets runtime despite a reduction in hit counts achieved in the bottom part of the function. Revisiting `cProfile` results, it becomes evident that all FINd functions henceforth collectively accounted for less than 50% of overall execution time (Figure 4, Subplot 2) which hints at newly accrued overhead in result of introduced *Numpy* code. A closer examination verified the emergence of unprecedented sub-function calls raking right below boxFilter in terms of `tottime` *(Appendix 2)*.

To reason this seemingly counterintuitive finding, it is important to understand how the original `boxFilter` function operates on its arguments. Aside from executing manageable amounts of rather trivial algebra, `boxFilter` heavily relies on repeat object indexing to route pixel elements between input and output arrays. While pure Python lists are in many ways inferior to *Numpy* arrays, they are well-suited to guarantee the kind of rapid element accesses in demand by `boxFilter` (Lanaro, 2000). In addition, list object creation within FINd's `fromImage` function is contingent on thumbnailing which anticipates taxing scenarios like memory reallocations (*O(N)*) and list insertions (*O(N)*) from happening (Lanaro, 2000). While lists carry pointers to where an object already exists in memory, *Numpy* arrays do not constitute object collections and require the construction of new python objects detached from the original array in response to an indexing operation (stackoverflow, 2015). By placing *Numpy* functions within those loops that could not be dismantled, attempted optimizations failed to use *Numpy* to its advantage (i.e. broadcasting) while introducing taxing substitutions at points where pure Python proves effective in consideration of the aforementioned particularities. Yet, well-intentioned enhancements using *Numpy* furnished *Numba* a fit occasion to streamline execution through compilation *(Figure 4, Subplot 3)*.

4 Part II

Part 2 revolves around the comparison of a performance optimised version of `FINd_base` against two alternative hashing approaches from the *imagehash* library. FINd optimization variant `numpy_numba` kept pace with `numpy_scipy` regarding overall execution time *(Figure 3)* while demonstrating less within-group hash variability as indicated by comparatively smaller error bars across image groups *(Figure 2, Subplot 2)*. `numpy_numba` was hence admitted to the comparative exercise.

Approaches Average Hash & Difference Hash (henceforth referred to as `ahash` and `dhash`) were selected from the *imagehash* library. Both algorithms share common traits in respect to image pre-processing (i.e. upfront image size reduction, greyscale conversion) and default output format (8x8 binary hash key) but apply different criteria to hash construction. While `ahash` compares all pixels against the average, `dhash` assigns hash bits based on differences between two horizontally adjacent pixels (Krawetz, 2013).

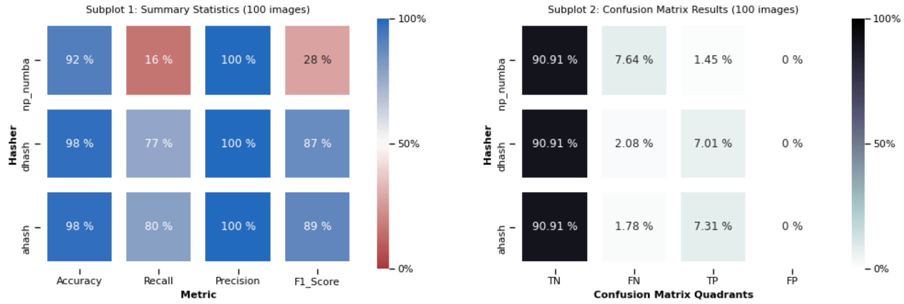
The report proceeds with a comparative analysis with focus on accuracy and performance and concludes with a discussion on the perceived utility of scrutinized approaches for *FIN*'s use case.

4.1 Comparative Analysis

An image hashing approach is deemed accurate when pairwise comparisons of hashes pertaining to similar images result in smaller hamming distances while yielding larger distances for distinct images. In favour of robust performance estimates, the assessment was grounded on a stratified random sample of 100 images across 10 groups, allowing for 4950 unique pairwise hash comparisons. Since previous studies found perceptual hashes to be prone to noise intake when larger hash sizes are prescribed (Jablons, 2017), default bit-string lengths of `ahash` and `dhash` (64 bit) were not altered to match the static bit size of `numpy_numba` (256 bit). To ensure comparability despite aforementioned differences, hamming distances were normalized against hash lengths produced by each algorithm. Following Krawetz's (2011) conventions, a uniform dissimilarity threshold of 16% was established below which a pair of images is qualified similar.

Confusion matrix results portray `ahash` and `phash` as high precision, high recall classifiers awarded with high F1-scores of 89% and 87% respectively *(Figure 5, Subplot 1)*. While FINd optimization variant `numpy_numba` performed on par in terms precision, a strikingly low recall rate of 16% reveals its inability to catch all true image similarities at the chosen threshold.

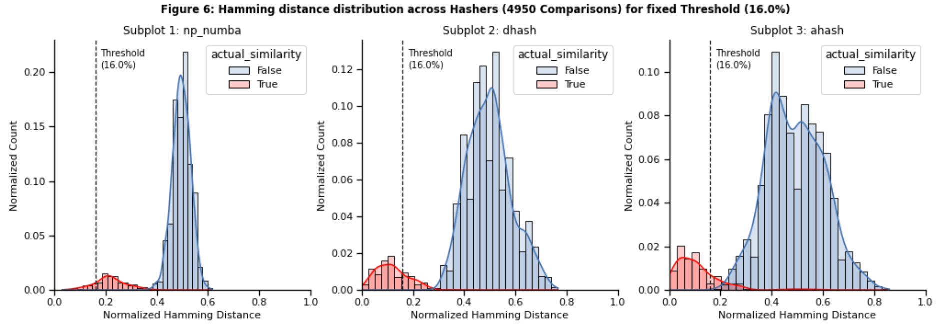
Figure 5: Accuracy Appraisal Results across Hashers (100 images / 4950 Comparisons)



This manifests itself in a heightened share of False negatives relative to `ahash` and `dhash` *(Figure 5, Subplot 2)* and eventually translates into a lower F1-Score of 28% *(Fig 5, Subplot 1)*. `numpy_numba`'s ability to yield high accuracy despite poor recall as well as disproportional amounts of true negatives in the confusion matrix across hashers *(Figure 5, Subplot 2)* further hints at a true class label distribution strongly skewed towards pairwise dissimilarities which stems from small-scale sampling across multiple image strata *(Appendix 3)*.

Yet, a series of histograms overlaying normalized hamming distance distributions with actual pairwise image similarities *(Figure 6, Subplots 1-3)* somewhat relativize the confusion matrix verdict.

Figure 6: Hamming distance distribution across Hashers (4950 Comparisons) for fixed Threshold (16.0%)



Two important findings emerge:

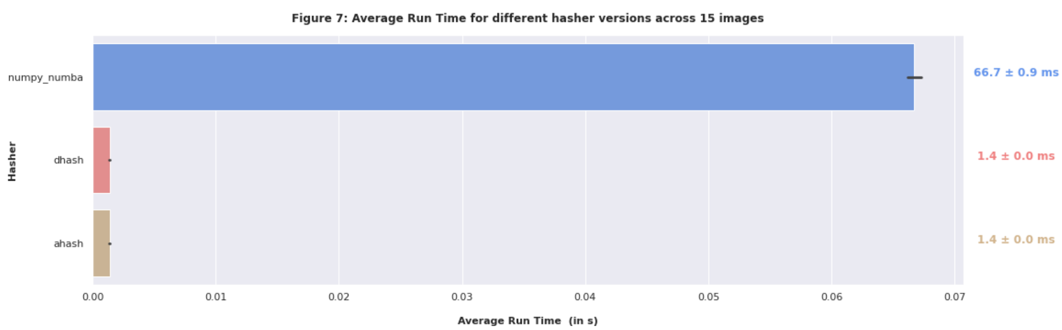
First, more broadly, quasi-symmetrically distributed hamming distances for truly similar instances suggest that `numpy_numba` is more variant to slight input changes but not necessarily less suitable to master the task at hand. Quite the contrary, non-overlapping class label distributions *(Figure 6, Subplot 1)* underpin `numpy_numba`'s ability to more effectively insulate actual matches from actual mismatches relative to `dhash` and `ahash` in particular. While `ahash` is comparatively invariant as indicated by a right-skewed true positive class label distribution *(Figure 6, Subplot 3)*, overlapping histograms allude to the emergence of false positives if the threshold was moved further to the right.

This brings to the second important take-away. Different hash approaches do not share the same optimal threshold. If the threshold was shifted from 16% up to 39%, `numpy_numba` would gain in recall without jeopardising precision and henceforth perform on par with competing *imagehash* approaches. *Table 3* summarizes changes in performance metrics associated with different threshold values for `numpy_numba`.

Threshold	Accuracy	Recall	Precision	F1_Score
16%	0.92	0.16	1	0.28
25%	0.98	0.75	1	0.86
35%	1	0.97	1	0.99
39%	**1**	**1**	**0.98**	**0.99**
40%	0.99	1	0.93	0.96
50%	0.47	1	0.15	0.25
60%	0.09	1	0.09	0.17

Table 3: *Hash performance across different hamming distance thresholds - numpy_numba*

With respect to computational efficiency, averaged program execution times reveal that `dhash` (~1.4 ms) and `ahash` (~1.4 ms) run more than 48-times faster than `numpy_numba` (66.7 ms) *(Figure 7)*.



Observed differences can be reasoned in light of prominent architectural differences. By computing "local averages" during boxFilter convolution followed by a "global average" computation on discrete cosine transformed (DCT) values, `numpy_numba` sets forth a much more sophisticated, yet computationally taxing workflow relative to its imagehash opponents. Both, `ahash` and `dhash` forgo any DCT and boxFilter convolutions (Krawetz, 2011; Krawetz, 2012) which were previously found to absorb ~41 ms and ~19 ms of `Numpy_Numba`'s execution time respectively *(Figure 4, Subplot 3)*.

Eventually, exemplary `cProfile` results *(Table 4 and 5)* show minuscule function runtimes for `ahash` and `dhash` confirming our suspicion of extremely light-weight computational routines underlying these hashes.

```
      230 function calls in 0.002 seconds

Ordered by: internal time
List reduced from 74 to 5 due to restriction <5>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.001    0.001 {method 'decode' of 'ImagingDecoder' objects}
     1    0.000    0.000    0.000    0.000 {method 'resize' of 'ImagingCore' objects}
     1    0.000    0.000    0.000    0.000 {built-in method io.open}
    38    0.000    0.000    0.000    0.000 {method 'read' of '_io.BufferedReader' objects}
     1    0.000    0.000    0.000    0.000 {method 'convert' of 'ImagingCore' objects}
```

Table 4: *Exemplary cProfile results for one image - dhash*

```
      243 function calls in 0.002 seconds

Ordered by: internal time
List reduced from 83 to 5 due to restriction <5>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.001    0.001 {method 'decode' of 'ImagingDecoder' objects}
     1    0.000    0.000    0.000    0.000 {method 'resize' of 'ImagingCore' objects}
     1    0.000    0.000    0.000    0.000 {built-in method io.open}
     1    0.000    0.000    0.000    0.000 {method 'convert' of 'ImagingCore' objects}
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
```

Table 5: *Exemplary cProfile results for one image - ahash*

A glance at the underlying function code on *GitHub* reveals that both *imagehash* algorithms shrink incoming images to the desired output hash length right at the start of the workflow (Buchner, 2020) whereas *FIN* 's implementation does not only accommodate larger inputs thanks to a more generous default thumbnailing threshold but also executes any computationally taxing functions on larger size intermediate arrays (64x64 & 64x16) in order to arrive at comparatively larger 256 bit-sized output hashes. The aforementioned characteristics – likewise inherited by `numpy_numba` - might contribute to more variant hash representations but eventually go on the expense of computational efficiency.

4.2 Discussion

In light of the previous analysis outcomes, the final part of this report opens with the clichéd but adequate notion of "it depends".

Performance appraisals portrayed `ahash` and `dhash` as computationally superior. Yet, accuracy appraisals have cautioned us not to jump to premature conclusions considering FINd variant `numpy_numba`'s potential to yield highly competitive results provided the right hamming distance threshold.

Eventually, evaluation results are indicative in that they draw from a particular breed of images but also recourse to a fraction of amenable profiling techniques. Therefore, subsequent points aim to contextualize analysis outcomes in considerations such as available IT infrastructure, anticipated image volume and complexity as well as targeted product /service scope. These factors are collectively going to frame *FIN* 's yet vaguely articulated ambition to match and cluster images at scale.

If *FIN* has limited computing resources at its disposal, `ahash` and `dhash` represent economically attractive choices to achieve high image turnover despite computational constraints given their miniscule per-image runtimes with memory footprint indistinguishable from `FINd_base` *(Appendix 4)*.

On the other hand, if *FIN* 's primary use case has low fault tolerance towards both, false positives and negatives, `numpy_numba` sets forth a sound implementation given its ability to neatly insulate pairwise similarities and dissimilarities *(Figure 6, Subplot 1)*. For instance, Copyright Infringement detection applies hashing to trigger legal action if a document hash is similar to a blacklisted item (Kumparak, 2014). Yet, cases of unjustified lawsuits in consequence of a false positive or unnoticed copyright breeches in consequence of false negatives are scenarios that have to be avoided at all cost.

Beyond that, some hashes are more qualified to handle higher content complexity than others.

While `ahash` is found to deteriorate in performance when facing routine transformations such as gamma corrections, `dhash` was found to be more robust in such cases (Krawetz, 2013). If *FIN* plans to perform image hashing to filter out sensitive or malicious content, purpose-built algorithms such crop-resistant hashing (Steinebach et al., 2014) might be even more suitable to deal with adversarial transformations like cropping, rotations, mirroring, bordering or noise found to test the wits of *imagehash* library implementations (Jablons, 2017).

Eventually, analysis results reviews across Part 1 and Part 2 meant to show that there is no free lunch. This motivates our recommendation to experiment with combinations of hashes as opposed to artificially restraining oneself to one vs. the other option. All hashes were found to run at a fraction of a second while yielding reliable classifications provided an adequate threshold was set. In result, a sound game plan would be to ground an image similarity detection engine on a majority voting mechanism which takes the judgement of each approach during *FIN* 's matching and clustering exercise into account. All things equal, if *FIN* should ever wish to cycle back from production to ideation, we would like to close this report with a friendly reminder that pure Python, while suffering from several drawbacks at scale, provides an uncontested environment for rapid prototyping.

5 Appendices

5.1 Appendix 1: Line Profiler results output for functions boxFilter and fillFloatLumaFromBufferImage for different FINd optimization variants

```
Function: fillFloatLumaFromBufferImage at line 164

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   164                                           def fillFloatLumaFromBufferImage(self, img, luma, numCols, numRows):
   165                                                   #R - type conversion to array
   166         1       1400.0   1400.0     33.5          rgb_image = np.asarray(img.convert("RGB"), dtype="float16")
   167                                                   #R - for loop replacement with matrix multiplication
   168                                                   #R - output formatting via unraveling & flooring
   169         1       2775.0   2775.0     66.5          return np.floor(np.ravel(np.dot(rgb_image[:][:], [0.299, 0.587, 0.114])))


Function: boxFilter at line 356

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   356                                           @classmethod
   357                                           #@jit
   358                                           def boxFilter(cls, inpt, output, rows, cols, rowWin, colWin):
   359                                                   """
   360                                                   My comments: spatial domain linear filter in which each pixel in the resul
   361                                                   the average value of its neighboring pixels in the input image; OPTIMIZATI
   362                                                   """
   363                                                   #R - converted back to list to render function input more ameanable to ope
   364         4       8039.0   2009.8    100.0          return cls.convolve_avg(inpt.reshape(rows, cols),
   365         3          3.0      1.0      0.0          rowWin,colWin).reshape(1, rows * cols)[0].tolist()
```

Table 1: *Line Profiler Results for fillFloatLumaFromBufferImage and boxFilter - Optimization variant numpy_scipy*

```
File: /Users/maximilianfaschan/Documents/GitHub/DAS/Summative/FINd_np.py
Function: fillFloatLumaFromBufferImage at line 163

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   163                                           def fillFloatLumaFromBufferImage(self, img, luma, numCols, numRows):
   164                                                   #R - type conversion to array
   165         1       1666.0   1666.0     32.9          rgb_image = np.asarray(img.convert("RGB"), dtype="float16")
   166                                                   #R - for loop replacement with matrix multiplication
   167                                                   #R - output formatting via unraveling & flooring
   168         1       3393.0   3393.0     67.1          return np.floor(np.ravel(np.dot(rgb_image[:][:], [0.299, 0.587, 0.114])))


Function: boxFilter at line 356

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   356                                           @classmethod
   357                                           #@timer
   358                                           #@profile
   359                                           #@jit
   360                                           def boxFilter(cls,input,output,rows,cols,rowWin,colWin):
   361         1          1.0      1.0      0.0          halfColWin = int((colWin + 2) / 2)  # 7->4, 8->5
   362         1          1.0      1.0      0.0          halfRowWin = int((rowWin + 2) / 2)
   363                                                   #R - Loop over range adapted to numpy
   364       251        165.0      0.7      0.0          for i in np.arange(0,rows):
   365                                                           #R - Loop over range adapted to numpy
   366     62750      36135.0      0.6      0.7                  for j in np.arange(0,cols):
   367                                                                   #C - func replacement with numpy alternative found to be c
   368     62500      87938.0      1.4      1.8                          xmin=max(0,i-halfRowWin)
   369     62500      56076.0      0.9      1.2                          xmax=min(rows,i+halfRowWin)
   370     62500      62897.0      1.0      1.3                          ymin=max(0,j-halfColWin)
   371     62500      52919.0      0.8      1.1                          ymax=min(cols,j+halfColWin)
   372     62500      27693.0      0.4      0.6                          s=0
   373                                                                   #R - loop compression by accessing entire column range at
   374    435250     558560.0      1.3     11.5                          for k in np.arange(xmin,xmax):
   375                                                                           #R -  numerical operations adapted to numpy
   376    372750    3545672.0      9.5     72.9                                  s += np.sum(input[(k*rows+ymin):(k*rows+ymax)])
   377                                                                   #R - loop compression by accessing entire column range at
   378                                                                   #R -  numerical operations adapted to numpy
   379     62500     437022.0      7.0      9.0                          output[i*rows+j] = np.true_divide(s,np.multiply((xmax-xmir
```

Table 2: *Line Profiler Results for fillFloatLumaFromBufferImage and boxFilter - Optimization variant numpy*

5.2 Appendix 2: Unprecendeted function overhead following pure Numpy optimization

```
Ordered by: internal time
List reduced from 117 to 5 due to restriction <5>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    1.222    1.222    3.655    3.655 FINd_np.py:356(boxFilter)
372750    0.828    0.000    0.828    0.000 {method 'reduce' of 'numpy.ufunc' objects}
372750    0.362    0.000    1.365    0.000 fromnumeric.py:73(_wrapreduction)
372750    0.350    0.000    1.770    0.000 fromnumeric.py:2092(sum)
372750    0.192    0.000    2.162    0.000 <__array_function__ internals>:2(sum)
```

5.3 Appendix 3: True pairwise similarity distribution

Metric	Values
# Groups sampled	**10**
# Sampled Images per Group	**10**
# of True unique pairwise similarities (%)	**450 (9%)**
# of True unique pairwise similarities (%)	**4500 (91%)**

5.4 Appendix 4: Comparative Memory Profiling

Hash	Peak Memory	Increment
FINd_base	205.98 MiB	0.00 MiB
numpy_numba	205.98 MiB	0.00 MiB

Hash	Peak Memory	Increment
dhash	205.98 MiB	0.00 MiB
ahash	205.98 MiB	0.00 MiB

6 References

Buchner, J. (2020)._imagehash_ [GitHub Repository].https://github.com/JohannesBuchner/imagehash

Daniel, J. (2012). *Choosing the type of probability sampling. In Sampling essentials: Practical guidelines for making sampling choices* .(125-174). SAGE Publications, Inc., https://www.doi.org/10.4135/9781452272047

Jablons, Z. (2017, May 31).*Evaluating Perceptual Image Hashes at OkCupid*.tech.okcupid.com.https://tech.okcupid.com/evaluating-perceptual-image-hashes-okcupid/

Krawetz, N. (2011, May 26).*The Hacker Factor Blog: Looks Like It*.hackerfactor.com. http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html

Krawetz, N. (2013, January 21).*The Hacker Factor Blog: Kind of Like That*.hackerfactor.com. http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html

Kumparak, G. (2014, March 30).*How Dropbox Knows When You're Sharing Copyrighted Stuff (Without Actually Looking At Your Stuff)*.techcrunch.com.https://techcrunch.com/2014/03/30/how-dropbox-knows-when-youre-sharing-copyrighted-stuff-without-actually-looking-at-your-stuff/

Lanaro, G. (2000). Python High Performance: Build robust application implementing concurrent and distributed processing techniques. Packt Publishing (Second Edigion)

Numba.pydata.org (2020a, December 31).*Numba: Numba makes Python code fast.* numba.pydata.org.http://numba.pydata.org

Numba.pydata.org (2020b, December 31).*Automatic parallelization with @jit.* numba.pydata.org.https://numba.pydata.org/numba-doc/latest/user/parallel.html#automatic-parallelization-with-jit

SciPy.org (2020a, December 31). *Frequently Asked Questions*.sciPy.org.com. https://www.scipy.org/scipylib/faq.html#how-can-scipy-be-fast-if-it-is-written-in-an-interpreted-language-like-python

SciPy.org (2020b, December 31). *scipy.signal.convolve2*.sciPy.org.com. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html

stackoverflow (2018).*Using line_profiler with numba jitted functions*.stackoverflow.com. https://stackoverflow.com/questions/54545511/using-line-profiler-with-numba-jitted-functions

stackoverflow (2017).*numba @jit slower that pure python?*.stackoverflow.com. https://stackoverflow.com/questions/45155781/numba-jit-slower-that-pure-python

stackoverflow (2015).*Numpy individual element access slower than for lists*.stackoverflow.com. https://stackoverflow.com/questions/29281680/numpy-individual-element-access-slower-than-for-lists

Steinebach, M., Liu, H., & Yannikos, Y. (2014). Efficient cropping-resistant robust image hashing. Proceedings - 9th International Conference on Availability, Reliability and Security, ARES 2014, 579–585. https://doi.org/10.1109/ARES.2014.85

Vanderplas, J. (2016). *Python Data Science Handbook.* O'Reilly Media, Inc.https://learning.oreilly.com/library/view/python-data-science/9781491912126/

[10]:
```
# Liberally drawing on a combination of code from:
# https://gist.github.com/agounaris/5da16c233ce480e75ab95980831f459e
# and
# https://stackoverflow.com/a/52187331

from notebook import notebookapp
import urllib
import json
import os
import ipykernel
import io
from IPython.nbformat import current

def notebook_path():

    connection_file = os.path.basename(ipykernel.get_connection_file())
    kernel_id = connection_file.split('-', 1)[1].split('.')[0]

    for srv in notebookapp.list_running_servers():
        try:
            if srv['token']=='' and not srv['password']:
                req = urllib.request.urlopen(srv['url']+'api/sessions')
            else:
                req = urllib.request.urlopen(srv['url']+
                    'api/sessions?token='+srv['token'])

            sessions = json.load(req)
            for sess in sessions:
                if sess['kernel']['id'] == kernel_id:
                    return os.path.join(srv['notebook_dir'],
                                        sess['notebook']['path'])
        except:
            pass  # There may be stale entries in the runtime directory
    return None

with io.open(notebook_path(), 'r', encoding='utf-8') as f:
    nb = current.read(f, 'json')

word_count = 0
for cell in nb.worksheets[0].cells:
    if cell.cell_type == "markdown":
```

```
        if cell['source'][:12] == "# References":
            continue
        else:
            word_count += len(cell['source'].replace('#','')
                                 .lstrip().split(' '))

print(f"The word count excluding Tables, Appendices, References {word_count -␣
 ↪490}")
#### PLEASE NOTE ####
# Reference Count is automatically precluded from count
# Words attributable to Tables & Appendices (490) were separatly counted and␣
 ↪subsequently substracted
```

```
The word count excluding Tables, Appendices, References 3463
```