Oxford Internet Institute, University of Oxford

Assignment Cover Sheet

Candidate Number *Please note, your OSS number is NOT your candidate number*	1047951
Assignment *e.g. Online Social Networks*	Data Analytics at Scale
Term *Term assignment issued, e.g. MT or HT*	MT
Title/Question *Provide the full title, or If applicable, note the question number and the FULL question from the assigned list of questions*	-
Word Count	3444

By placing a tick in this box ☒ I hereby certify as follows:

(a) This thesis or coursework is entirely my own work, except where acknowledgments of other sources are given. I also confirm that this coursework has not been submitted, wholly or substantially, to another examination at this or any other University or educational institution;

(b) I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at https://www.ox.ac.uk/students/academic/guidance/skills?wssl=1.

(c) I agree that my work may be checked for plagiarism using Turnitin software and have read the Notice to Candidates which can be seen at: http://www.admin.ox.ac.uk/proctors/turnitin2w.shtml and that I agree to my work being screened and used as explained in that Notice;

(d) I have clearly indicated (with appropriate references) the presence of all material I have paraphrased, quoted or used from other sources, including any diagrams, charts, tables or graphs.

(e) I have acknowledged appropriately any assistance I have received in addition to that provided by my [tutor/supervisor/adviser].

(f) I have not sought assistance from a professional agency;

(g) I understand that any false claims for this work will be reported to the Proctors and may be penalized in accordance with the University regulations.

Please remember:

- To attach a second relevant cover sheet if you have a disability such as dyslexia or dyspraxia. These are available from the Higher Degrees Office, but the Disability Advisory Service will be able to guide you.

Submission for Data Analytics at Scale
Report for Find Images Now (FIN)

Candidate number: 1047951

January 2021

Abstract

The problem FIN is facing is CPU-bound. In order to improve the performance of the FINd algorithm, three optimizations were made: Cython (14.3 ms per image), CV2 and Numpy (44.6 ms per image), and Spark (benefits from processing a larger number of images). The Cython FINd implementation was also compared to Perception and Average hashing. Although the speed of the FINd algorithm is slower, it offers more accurate hashing. The decision regarding which algorithm (and implementation) to use depends on the intended use-case of FIN.

1 Part One

In this section, I perform (a) an initial assessment of the FINd performance in terms of time and memory on both program and function levels; (b) an assessment of its scalability; and (c) perform three optimizations on FINd. The performance ('accuracy') of the algorithm is described in Part 2 relative to two other hashing algorithms, together with an elaborate explanation of the methodology employed.

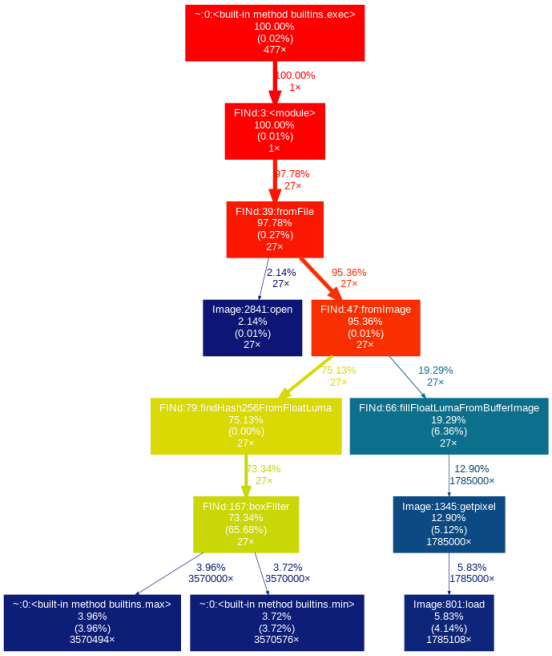
1.1 FINd analysis

1.1.1 Initial FINd runtime assessment

Running the hashing algorithm on a single example takes 530 ms, ± 10 ms (mean ± standard deviation) on Oxford's Linux server, as measured by wall-clock time. The average CPU time is around 520 ms. The CPU/seconds ratio is approximately equal to one for a single processor and one image, indicating that this is likely a CPU-bound problem.

The breakdown of the time required for each part of the algorithm is provided in a dot graph in Figure 1 (developed using $gprof2dot$).

Figure 1: Initial profiling of the code



Only nodes that take up 2% or more time are shown. Two high-level functions take up the most time: *findHash256FromFloatLuma* (75% of the time) and *fillFloatLumaFromBufferImage* (19% of the time). To optimize the former, changing the function *boxFilter* is required. The latter is mostly inefficient because of the Image *getpixel* method. These two functions are profiled in the next section.

1.1.2 Runtime assessment on a function level

The breakdown of the first function, *fillFloatLumaFromBufferImage*, is presented below.

Figure 2: First function line profiling

```
Total time: 0.268825 s
File: FINd_profile.py
Function: fillFloatLumaFromBufferImage at line 66

Line no.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    66                                           @profile
    67                                           def fillFloatLumaFromBufferImage(self, img, luma):
    68         1          3.0      3.0      0.0         numCols, numRows = img.size
    69         1        208.0    208.0      0.1         rgb_image = img.convert("RGB")
    70         1          1.0      1.0      0.0         numCols, numRows = img.size
    71       251        124.0      0.5      0.0         for i in range(numRows):
    72     62750      27512.0      0.4     10.2             for j in range(numCols):
    73     62500     184795.0      3.0     68.7                 r, g, b = rgb_image.getpixel((j, i))
    74                                                       luma[i * numCols + j] = (
    75                                                           self.LUMA_FROM_R_COEFF * r
    76                                                           + self.LUMA_FROM_G_COEFF * g
    77     62500      56182.0      0.9     20.9                 + self.LUMA_FROM_B_COEFF * b
    78                                                       )
```

The most significant amount of time is attributed to the *.getpixel* method, as already suggested by the initial overview. We further see that the for loop takes an additional 10% of the time. Line profiling for the second function is provided below.

Figure 3: Second function line profiling

```
Total time: 3.30221 s
File: FINd_profile.py
Function: boxFilter at line 168

Line no.    Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   168                                           @classmethod
   169                                           @profile
   170                                           def boxFilter(cls,input,output,rows,cols,rowWin,colWin):
   171         1          2.0      2.0      0.0         halfColWin = int((colWin + 2) / 2)
   172         1          1.0      1.0      0.0         halfRowWin = int((rowWin + 2) / 2)
   173       251        111.0      0.4      0.0         for i in range(0,rows):
   174     62750      29737.0      0.5      0.9             for j in range(0,cols):
   175     62500      30148.0      0.5      0.9                 s=0
   176     62500      45175.0      0.7      1.4                 xmin=max(0,i-halfRowWin)
   177     62500      42651.0      0.7      1.3                 xmax=min(rows,i+halfRowWin)
   178     62500      43619.0      0.7      1.3                 ymin=max(0,j-halfColWin)
   179     62500      42881.0      0.7      1.3                 ymax=min(cols,j+halfColWin)
   180    435250     228763.0      0.5      6.9                 for k in range(xmin,xmax):
   181   2595831    1383380.0      0.5     41.9                     for l in range(ymin,ymax):
   182   2223081    1407445.0      0.6     42.6                         s+=input[k*rows+l]
   183     62500      48294.0      0.8      1.5                 output[i*rows+j]=s/((xmax-xmin)*(ymax-ymin))
```

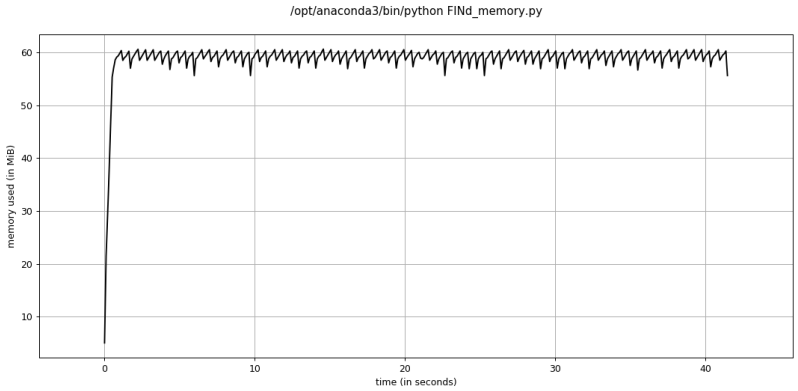
Note that most of the computation is being done within the two inner loops.

Looping through the last loop takes about 42% of the time and another 42% for the final slicing and addition.

1.1.3 Profiling for memory usage

A potential issue for computational performance might be accessing items stored in different memory locations. If a program has used up the working memory of a function, the execution will be slower, since files and objects are accessed through other memory, such as hard drives. This takes a substantially longer time. After running *memit* on a number of different images, the reported memory increments were constant, at around 0.8 - 2.7 MiB. This suggests that the usage of the memory is constant. Furthermore, Figure 4 provides the memory usage of hashing 72 images at once.

Figure 4: Profiling the memory usage of FINd

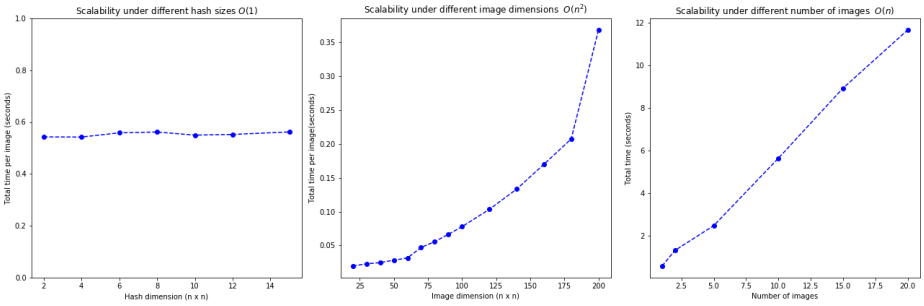


The memory usage is constant through time, independent of the number of images. The small oscillations around 60 MiBs represent the relative memory usage of hashing one image (in a similar range as reported with *memit*). This is because image hashing is a serial process — the function loops over the images, hashes an image, and returns the output before taking another input. Furthermore, each meme is relatively small and does not require much RAM for keeping it in memory. The distribution of memory usage for the key functions is provided in Appendix A. Memory usage mostly augments in the for loops of several functions. Generally, FINd does not appear to be facing a memory problem.

1.2 Scalability

Figure 5 depicts the computational complexity of: (a) changing the hash output dimensions, $O(1)$; (b) changing the input image dimensions, $O(n^2)$; and (c) changing the number of images, $O(n)$.

Figure 5: Scalability under different parameters



In case FIN would like to increase the processing speed, resizing the image is the best alternative that could be applied to the actual image. Note that this is likely to lead to a substantial drop in the hashing algorithm's ability to differentiate between groups. Without any multiprocessing solution (as introduced in optimization 3), the scale is linear, requiring about 8 hours to process the existing data set. Notice that the hash size does not affect the speed (as I discuss in Part 2, the other two algorithms are faster but not as a result of smaller hashes).

1.2.1 Initial analysis concluding remarks

Two functions are identified as bottlenecks: *boxFilter* and *fillFloatLumaFromBufferImage*. Memory does not appear to be an issue for the program. Furthermore, there is superfluous code (such as creating the same variables multiple times or unnecessarily copying variables) that hinders the readability and marginally affects performance. Standards such as PEP-8 are not followed. This does not affect performance per se but might affect the speed at which other engineers/scientists develop this code in the future. The optimized code resolves some of these issues and provides a solid foundation for future work.

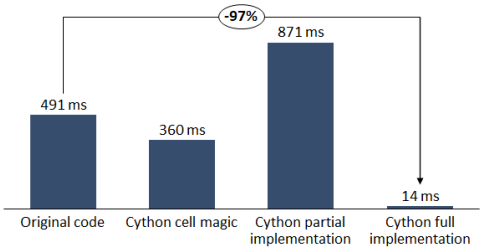
1.3 Optimizations

In this section, I optimize the code with three approaches: Cython, CV2/numpy, and Spark.

1.3.1 Cython

Cython is a language that extends Python by supporting the declaration of types for functions, variables, and classes, which, in turn, allows Cython to compile Python to C code. Cython is a good candidate for achieving optimizations, as the initial code is written in pure Python. The goal of the optimization is to remove all possible interactions with Python to get the maximum speed-up. The results of the optimizations are provided in Figure 6.

Figure 6: Cython computational times.



The figure provides four numbers: the timing of the original code, using Cython with just the cell magic, implementing a partial improvement for one function, and converting all the code into Cython, yielding a time improvement of 97% per image. The first optimization, using just the cell magic, offers minimal speedup, as Python is actively invoked in most of the program. After performing initial optimizations, the run time increased. This is likely caused by the increased overhead of switching between C and Python with only minimal reduction in speed from defining the variables. To make use of Cython, the whole program was optimized. After declaring all the variable types, memory-view objects were created for each variable, enabling Cython to access information quickly and decreasing the number of times Cython switches to Python. The two functions identified as having the largest overhead are almost entirely run in C with little-to-no interaction with Python. The computational complexities for the images are unchanged.

Given that the issue is CPU-bound, finding ways to improve the performance of a single image offers great benefits for the whole data set. Memory is not inspected, as it is not one of the issues identified, but the memory usage is likely to decrease, as Cython requires to explicitly assign memory for each variable. This offers an advantage over Python that is not aware of, for example, the size of a variable and might need to move memory around to perform an operation if the variable exceeds the memory allocated initially.

It is important to note the drawbacks of Cython. Any future improvements and developments of the FIND algorithm might take a longer time than in Python for two reasons. For one, Cython has less support than other packages, making it harder to troubleshoot issues. For another, it requires more planning in advance, given the need to type out data types or variables explicitly.

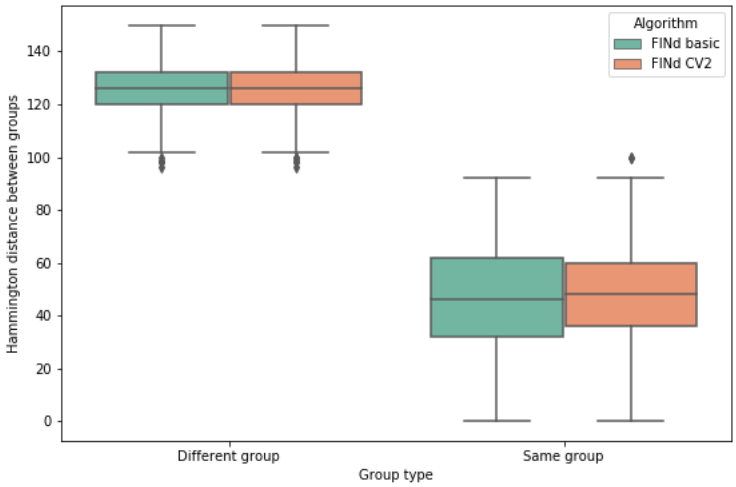
1.3.2 CV2 and Numpy

Another optimization was performed by using $CV2$ (or *opencv*) and $Numpy$. This reduced the speed per image to 44 ms by changing the *fillFloatLumaFromBufferImage*, *BoxFilter* functions together with other minor changes. Note, however, that the CV2 box filter solution changed the Hamming distances of the images. This is mostly a result of a different implementation of box filter. Whereas the FINd algorithm minimizes the box size used for the convolution operation at the edges (decreasing the box size even up to a $3x3$ dimension), in the CV2 implementation, an additional layer of white padding is added. The algorithm still uses average hashing with the same box dimensions.

However, the overall performance of the FINd algorithm is unlikely to affect

performance in a production environment. To understand why, refer to Figure 7.

Figure 7: Comparison of the Hamming distances between the CV2 and initial implementation



Note: The comparison was made between the images of 6 random meme families, where each meme family had 10 images assigned to it.

The figure depicts box plots for the Hamming distances for the initial FINd algorithm, and the new CV2-adjusted version, for both different and same group images. The overall differences between the groups have not changed. This is important, as, for hashing images, what matters is the relative similarity and difference of images from meme families. In this case, this relative similarity is unaltered, even if the hash of the original image is slightly different. Therefore, this should not affect overall performance in a production environment.

Apart from the speed boost, the code for CV2 is much easier to understand and implement. Under the hood, CV2 uses C++ which provides faster and more optimized computation than the initial pure Python implementation. One drawback of this implementation is that there is less flexibility to make specific changes to the hashing algorithm if such changes would be required.

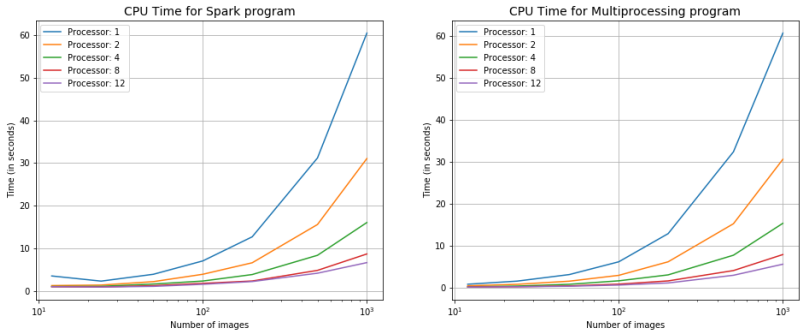
1.3.3 Spark

The last optimization performed is using Spark. The algorithm was not re-written in pyspark from scratch; a user-defined function (UDF) was created and the hashing algorithm was mapped onto a spark dataframe containing image paths.

Spark should enable faster performance as the number of images grows for multiple reasons. First, Spark uses in-memory processing. This means that there should be fewer read-and-write operations conducted which, in turn, should speed up the processing speed. Second, Spark works on a specific data structure called a resilient distributed dataframe (RDD) which partitions the data across the nodes of a cluster (e.g. multiple servers). Third, Spark enables parallel processing. Fourth, directed acyclic graphs (DAGs) are constructed that optimize calculations.

After performing the optimizations, the CPU time was measured for different numbers of processors. To understand the relative benefit of multiprocessing and other Spark-related features mentioned in the previous paragraph, a Python-based multiprocessing algorithm is used as a time benchmark. This is presented in Figure 8.

Figure 8: Comparison of the CPU times between Spark and Multiprocessing for different processors and number of images



Note: the CV2 implementation of the algorithm is used for Spark and multiprocessing. CPU time is measured.

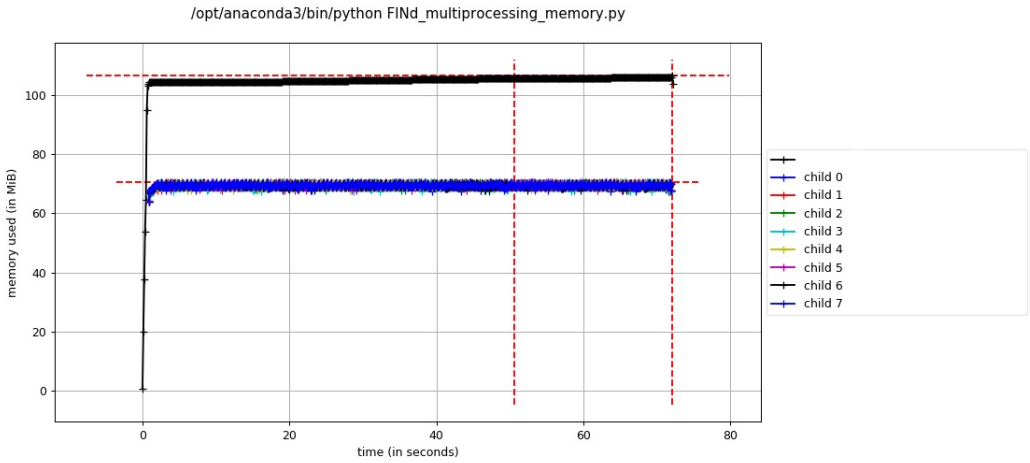
The left plot showcases the CPU time for Spark; the right indicated the same for multiprocessing. The figure suggests that the CPU usage is the same for Spark and multiprocessing. This means that Spark, in this case, is not able to utilize its RDDs, DAGs, in-memory processing, and other benefits to achieve any speedup for CPU time, likely because a UDF is not able to convert the processing from Python to a Spark engine. In order to reap the benefits of Spark, re-writing the algorithm in pyspark can be considered.

Using Spark offers several opportunities for further implementations. First, it can be combined with any existing implementation to utilize the power of multiprocessing. Second, Spark can be scaled to multiple nodes easily. In case FIN requires even faster computation, this enables distributed computing. Third, RDDs distribute their data across multiple nodes, allowing for efficient distributed and parallel and distributed processing. The overhead from multiprocessing (both in Python and Spark) comes from the creation of separate

tasks for each chunk, serialization of those tasks, and waiting for shared memory. However, these overhead costs are marginal relative to the performance gains as the number of images increase.

One drawback of this approach is that each process uses memory. Figure 9 depicts the memory usage of 8 processes for hashing 100 images.

Figure 9: Memory usage for multiprocessing



Note: 8 multiprocessors and 100 examples are used with the initial FINd algorithm

Note that the overall memory consumption for all the processes in the server is above 100 MiBs. Although this is higher than before, the scale of the processes is unlikely to cause memory issues.

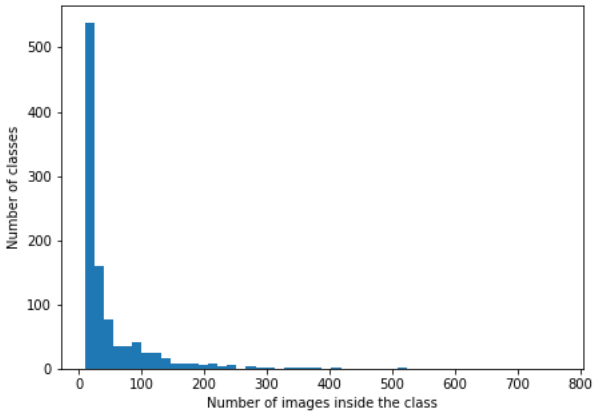
2 Part 2

In this section, I compare FINd's algorithm (in particular, the Cython implementation that provides the same output) with Perception Hashing and Average Hashing.

2.1 Number of classes

A method of comparison has to be established. For this, it is important to take note of the number of classes and images in each class. This distribution is illustrated in Figure 10.

Figure 10: Number of images per separate meme family

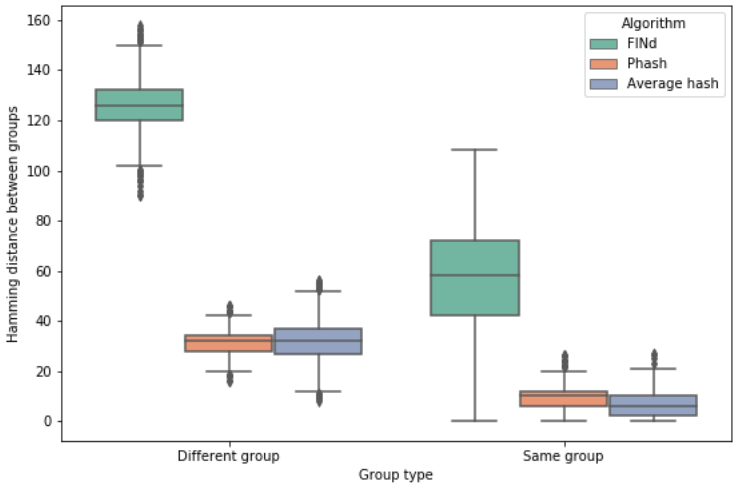


The figure showcases that the images approximate a power-law distribution, where most meme categories have 10 images and only a few have over 400. This is important for adjusting the performance measurement. In this report, I assume that classifying each group, as opposed to each image, is equally important. This implies that weighted/stratified sampling techniques are utilized (the details of which are discussed later). However, the average Hamming distance within each group is likely to be captured with a smaller sample size due to the relative similarities of the images.

2.2 Hamming distances of the algorithms

To calculate the Hamming distances of the image hashes produced by the three algorithms, I extract all the images in 12 meme families (where a meme family is, for instance, 0000_), and compare the Hamming distances between all the images. The distances between memes from the same and different groups are plotted separately and are illustrated in Figure 11.

Figure 11: Average Hamming distances of images from the same and different meme families

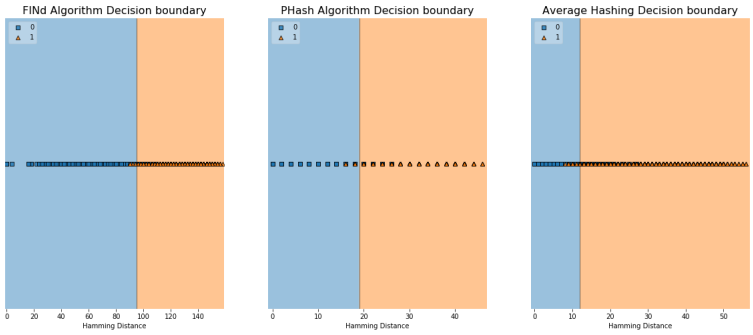


The Hamming distances are larger for the FINd algorithm in both group types. This is to a large extent because the size of the hash of the FINd algorithm is larger (16x16 instead of 8x8). This means that separate thresholds should be considered for all three algorithms.

2.3 Computing separate thresholds

In order to find the optimal thresholds for each group, I use a linear Support Vector Machine classifier on a subset of the data with the same difference measurements calculated in the previous section; the differences are labeled as either between the same meme family ($y = 0$) or different families ($y = 1$). The model is trained to identify which threshold best separates the two classes. This method is useful since only the support vectors (or, alternatively, differences close to the threshold) matter for finding the optimal threshold in one-dimensional space. The decision boundary for the three algorithms is presented in Figure 12.

Figure 12: Linear Support Vector Classifier for the distances between a sample of images for three hashing algorithms



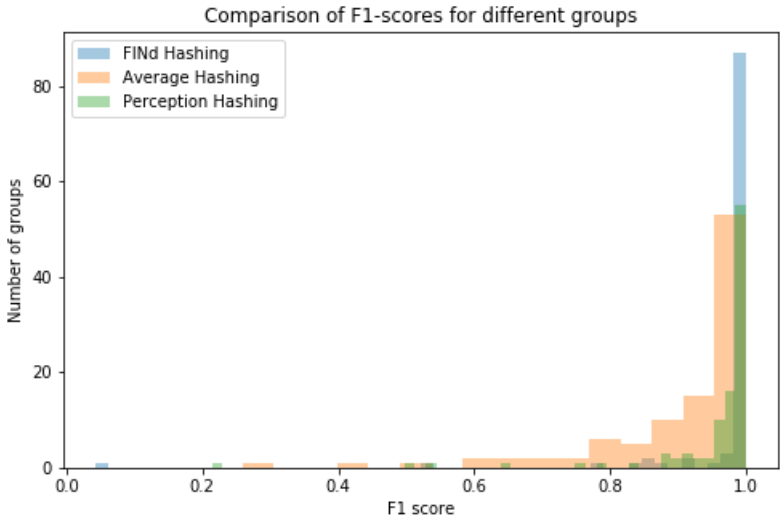
Note: The x-axis is different for all the images to showcase the relative distribution of the Xs around the separating hyperplane.

2.4 Measuring Performance

In this section, I compute the performance for each algorithm by using the macro-averaged F1-score. The calculation methodology is as follows. First, 100 random meme groups are selected with 10 different images from each group. Second, for each group of 10 images, the difference is calculated between the groups within that family, as well as 20 other randomly selected images from other meme families. This leads to a total of $\frac{30!}{2!(30-2)!} = 435$ image combinations (and distance measurements) for each group. Comparing the differences between other groups is important to measure the relative distance between the same and different meme families. Third, the F1-score is computed for the 435 difference measurements within that group. In this scenario, a false positive is a distance smaller than the specified threshold between memes from different classes; a false negative is a distance between two memes from the same class that is higher than the threshold. Lastly, all the F1-scores for the 100 groups are averaged to give a macro-F1 score, so that the F1-score for each group has the same weight.

The reason why the F1-score is chosen is that for each group the class is imbalanced: more images are from different groups than from the same group. Using accuracy can thus be misleading (e.g. small Hamming distances between all the images would give high accuracy). The reason why a macro (as opposed to micro) F1-score is used is to capture the relative performance of the hashing algorithm for each group. The distributions of the group F1-scores are illustrated in Figure 13.

Figure 13: F1-scores of 100 randomly sampled groups for three hashing algorithms



The graph showcases that FINd has most groups that have an F1-score of 100%; the other two algorithms show worse performance. The macro-averaged F1-scores are as follows.

- FINDHasher: 97.28%
- Average Hashing: 90.47%
- Perceptual Hashing: 95.26%

The single low F1-Score for FINd is a result of incorrect ground-truth data (discussed in the next paragraphs).

Depending on the use-case, however, the performance of these algorithms might be different. If FIN decides to use this algorithm for new incoming meme classification, it is sensible to classify the meme to the group with the lowest Hamming distance. However, given the large number of images, it is likely that another image will have a small distance just by chance.

To illustrate this, I run an additional simulation. One image from each group is picked representing the 'ground-truth' image for the group, resulting in a total of $n = 1035$ groups and images. Then a random meme from the remainder of 54,937 images is picked, and the Hamming distance is compared to the ground-truth memes. The meme is classified into the group with the shortest Hamming distance. After repeating this task 15,000 times (with replacement), the hashing algorithms classified the meme into the correct family the following number of times:

- FINDHasher: 85.23%
- Average Hashing: 71.3%
- Perceptual Hashing: 78.1%

This, on the one hand, illustrates the possible issue with classifying the images to a new family as they appear. On the other hand, the accuracy is relatively small because the 'ground-truth' data by FIN is not entirely accurate. Some images appear in certain groups more than once which makes the comparisons inaccurate. Examples of incorrectly classified images are provided in Appendix B. In the appendix, I showcase an image and the meme that represents the group it was (incorrectly) classified to. In many of these scenarios, the memes come from the same meme family but are labeled differently.

2.5 Measuring time and memory

The times were measured for all the hashing algorithms and are provided below:

- FINDHasher (in Cython): 14.6 ms
- Average Hashing: 1.74 ms
- Perceptual Hashing: 2.04 ms

FINDHasher in Cython is slower than the other two alternatives. Apart from changing the algorithm, an efficient way to optimize the speed would be to change the image quality. Changing the hashing output would not alter the speed of the algorithm.

Note that the memory usage of all the functions is the same, as described in Appendix C, and does not scale with more images.

2.6 Discussion on the hashes

The choice of the algorithm depends on the applications on FIN.

If speed is the priority for FIN, using Average hashing is preferred. The technique is mostly fast because it shrinks down the image to 8x8 and applies other basic transformations on the shrunken meme. Recall that reducing the image size changes the time complexity by $O(n^2)$ (Figure 5). It is called *average* since it simply takes the average pixel value, another fast operation. Average hashing is likely to be sufficiently useful and accurate for identifying identical images that have different image sizes, yet perform worse where more nuance is involved (which is reflected in its accuracy and F1-score).

Perception hashing is slower, yet more accurate (Figure 13), as it uses discrete cosine transforms to reduce frequencies (something also done by FINd Hashing). PHash is able to identify more detail and modifications to the text (such as differences in the text of the meme) more accurately. One of the

likely reasons why the algorithm achieves superior performance relative to average hashing is that it reduces the image size to 32x32 before doing any preprocessing.

FINDHasher is the slowest and most accurate of the three. One of the reasons it is slower is that the image with FINDHasher is reduced to 512x512 (although much of the original images have smaller dimensions) which requires more time to process the meme. The initial FINDHasher also uses other operations that consume time, such as changing the box filter size within 4 for-loops. The Cython FINd optimization is slower than average and perceptual hashing by 8.3 and 7.2 times for a single image, respectively. However, if accuracy is the priority, FINDHasher should be preferred.

FIN should decide on the relative costs of misclassification and processing time. For instance, to classify 10 million new incoming memes to a certain meme family (using the methodology described before), the total times and incorrectly classified images for the three algorithms would be as follows.

Table 1: Time and accuracy for classifying 10 million images

	Phash	AvgHashing	FINd
Time per image (ms)	2.04	1.74	14.6
Accuracy	78.10%	71.30%	85.23%
Total time (hours)	5.7	4.8	40.6
Incorrectly classified images (in thousands)	2190	2870	1477

It would take FINd 35.8 hours more to process 10 million mages on a single processor but it would lead to 1.39 million fewer misclassifications, assuming a similar data distribution. These numbers are pessimistic, however, since, as reported previously, the ground-truth data was not accurate. Scaling on multiple processors would reap the same benefits for all the algorithms. Ultimately, it is important to know the use-case.

3 Appendices

A Memory usage per function

Figure 14: Box Filter memory usage

```
Filename: FINd_memory.py

Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
   169   58.848 MiB   58.848 MiB           1   @classmethod
   170                                         @profile
   171                                         def boxFilter(cls,input,output,rows,cols,rowWin,colWin):
   172   58.848 MiB    0.000 MiB           1           halfColWin = int((colWin + 2) / 2)  # 7->4, 8->5
   173   58.848 MiB    0.000 MiB           1           halfRowWin = int((rowWin + 2) / 2)
   174   60.707 MiB    0.000 MiB         251           for i in range(0,rows):
   175   60.707 MiB    0.285 MiB       62750               for j in range(0,cols):
   176   60.707 MiB    0.000 MiB       62500                   s=0
   177   60.707 MiB    0.000 MiB       62500                   xmin=max(0,i-halfRowWin)
   178   60.707 MiB    0.020 MiB       62500                   xmax=min(rows,i+halfRowWin)
   179   60.707 MiB    0.000 MiB       62500                   ymin=max(0,j-halfColWin)
   180   60.707 MiB    0.094 MiB       62500                   ymax=min(cols,j+halfColWin)
   181   60.707 MiB    0.027 MiB      435250                   for k in range(xmin,xmax):
   182   60.707 MiB    0.832 MiB     2595831                       for l in range(ymin,ymax):
   183   60.707 MiB    0.547 MiB     2223081                           s+=input[k*rows+l]
   184   60.707 MiB    0.055 MiB       62500                   output[i*rows+j]=s/((xmax-xmin)*(ymax-ymin))
```

Figure 15: FromBufferImage memory usage

```
Filename: FINd_memory.py

Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
    67   56.508 MiB   56.508 MiB           1   @profile
    68                                         def fillFloatLumaFromBufferImage(self, img, luma):
    69   56.508 MiB    0.000 MiB           1           numCols, numRows = img.size
    70   56.766 MiB    0.258 MiB           1           rgb_image = img.convert("RGB")
    71   56.766 MiB    0.000 MiB           1           numCols, numRows = img.size
    72   58.809 MiB    0.000 MiB         251           for i in range(numRows):
    73   58.809 MiB    1.648 MiB       62750                   for j in range(numCols):
    74   58.809 MiB    0.328 MiB       62500                           r, g, b = rgb_image.getpixel((j, i))
    75                                                             luma[i * numCols + j] = (
    76                                                                     self.LUMA_FROM_R_COEFF * r
    77                                                                     + self.LUMA_FROM_G_COEFF * g
    78   58.809 MiB    0.066 MiB       62500                             + self.LUMA_FROM_B_COEFF * b
    79                                                             )
```

Figure 16: From File memory usage

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
    38   54.125 MiB   54.125 MiB           1    @profile
    39                                         def fromFile(self, filepath):
    40   54.125 MiB    0.000 MiB           1            img = None
    41   54.125 MiB    0.000 MiB           1            try:
    42   54.773 MiB    0.648 MiB           1                    img = Image.open(filepath)
    43                                               except IOError as e:
    44                                                       raise e
    45   58.082 MiB   58.082 MiB           1            return self.fromImage(img)
```

Figure 17: From Image memory usage

```
Filename: FINd_memory.py

Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
    47   54.773 MiB   54.773 MiB           1    @profile
    48                                         def fromImage(self,img):
    49   54.773 MiB    0.000 MiB           1            try:
    50                                                       # resizing the image proportionally to max 512px width and max 512px height
    51   55.477 MiB    0.703 MiB           1                   img=img.copy()
    52   55.477 MiB    0.000 MiB           1                   img.thumbnail((512, 512))
#https://pillow.readthedocs.io/en/3.1.x/reference/Image.html#PIL.Image.Image.thumbnail
    53                                               except IOError as e:
    54                                                       raise e
    55   55.477 MiB    0.000 MiB           1           numCols, numRows = img.size
    56   55.992 MiB    0.516 MiB           1           buffer1 = MatrixUtil.allocateMatrixAsRowMajorArray(numRows, numCols)
    57   56.508 MiB    0.516 MiB           1           buffer2 = MatrixUtil.allocateMatrixAsRowMajorArray(numRows, numCols)
    58   56.508 MiB    0.000 MiB           1           buffer64x64 = MatrixUtil.allocateMatrix(64, 64)
    59   56.508 MiB    0.000 MiB           1           buffer16x64 = MatrixUtil.allocateMatrix(16, 64)
    60   56.508 MiB    0.000 MiB           1           buffer16x16 = MatrixUtil.allocateMatrix(16, 16)
    61   56.508 MiB    0.000 MiB           1           numCols, numRows = img.size
    62   58.848 MiB   58.848 MiB           1           self.fillFloatLumaFromBufferImage(img, buffer1)
    63   58.848 MiB    0.000 MiB           1           return self.findHash256FromFloatLuma(
    64   60.832 MiB   60.832 MiB           1                   buffer1, buffer2, numRows, numCols, buffer64x64, buffer16x64, buffer16x16
    65                                               )
```

B Hashing algorithms and their misclassifications

Note: the Appendix showcases the misclassifications of 8 guess images relative to ground-truth images for three algorithms. Each image has a title with the full image name, the image number, and whether the image was the guess image (the random sampled image), or the ground-truth image. If the memes come from the same family, it indicates that there are two memes that are the same, yet from different meme families

Figure 18: Guess versus ground-truth data for Average Hashing

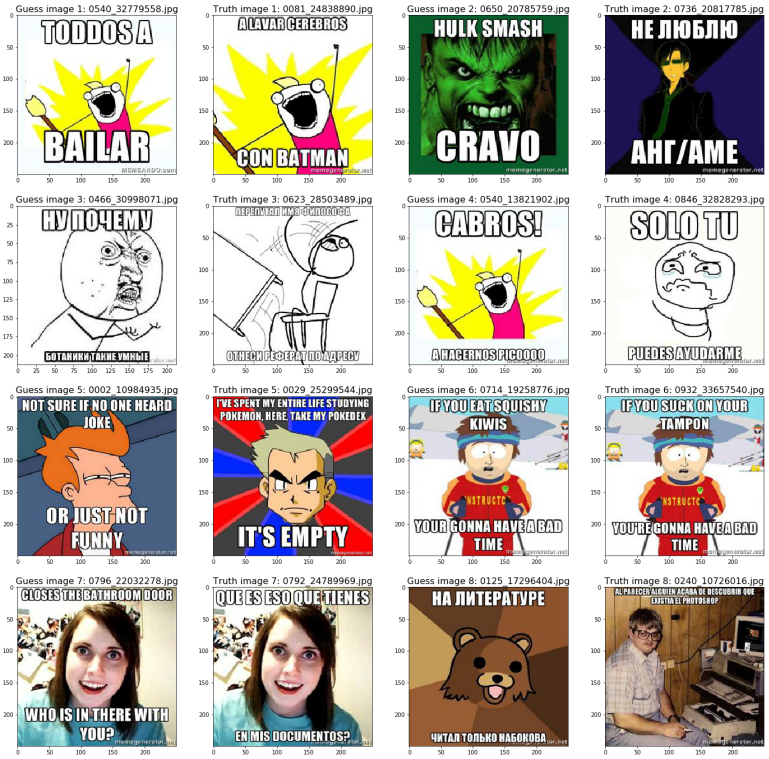
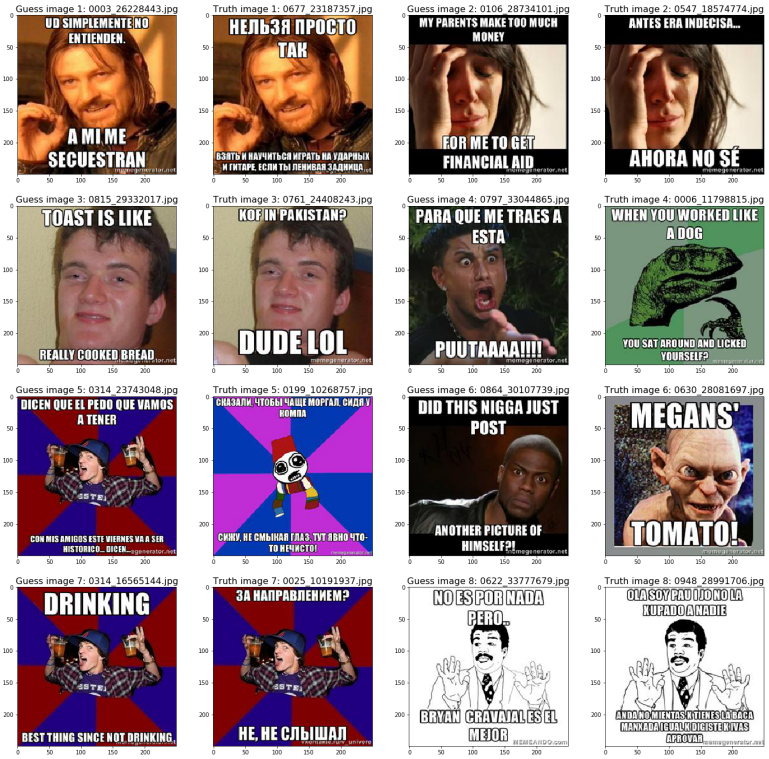


Figure 19: Guess versus ground-truth data for FINd Hashing

Figure 20: Guess versus ground-truth data for Perception Hashing



C Memory usage for hashing algorithms

Note: All the following measurements are done on a thousand images loaded in a pandas series.

Figure 21: Memory usage for the Average Hashing algorithm

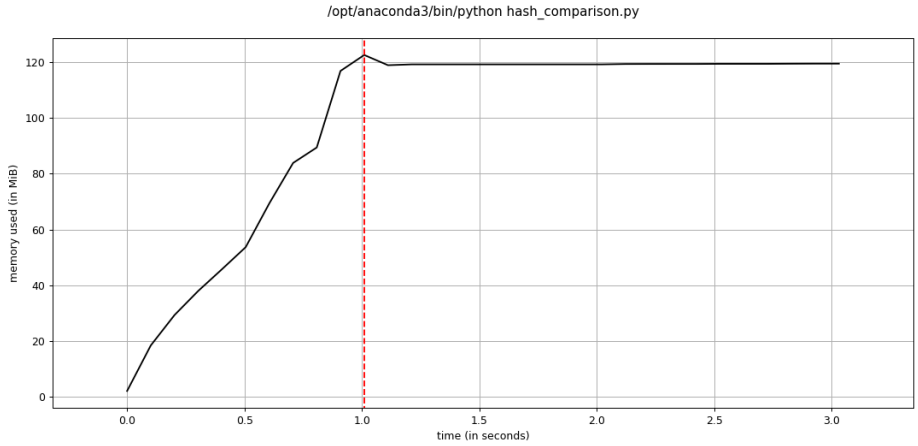


Figure 22: Memory Usage for the PHash algorithm

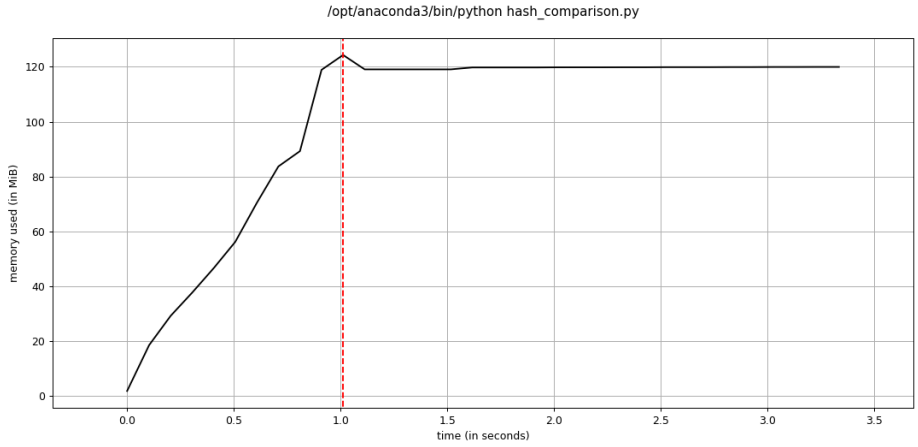


Figure 23: Memory usage for the Cython algorithm