Mention that I took the inspiration from the four methods from opencv. So I decided to manually implement each of them and compare the results to find the best performing model.

Test functions using RBG, then move to using HSV because its better

* Reference paper about why HSV is better than RGB
* Reference paper about why 80/20 split is a good ratio for tests

431 total images

20% test images (86.2)(86) – test folder

80% image gallery (344.8)(345) - gallery folder

4 image categories. Split test images accordingly.

86/4 = 21.5

22 crowds, 21 F1-cars, 22 horses, 21 landscapes

When testing I use the same test images for simple model

Why 15 returned images in better than initial 10 proposed

Why I chose to save the data I did

Why update database function is much better than calculating all image data every runtime

\*how I implemented the timer\*

\*how I calculated HSV array\*

\*sorting method\*

Generating HSV array for the query image every time is better because it doesn’t have to exist in the image database. Query time would be improved if I just identified it by it file path and took it from the read it from the test db but I felt that this would reduce the flexibility and real-world functionality too much that I decided to stick with generating the HSV array every time.

Increase heap space in Eclipse (-Xmx9216m)

Time each model

**Notes:**

431 total images

20% test images (86.2)(86) – test folder

80% image gallery (344.8)(345) - gallery folder

4 image categories. Split test images accordingly.

86/4 = 21.5

22 crowds, 21 F1-cars, 22 horses, 21 landscapes

Current precision is based on 15 images returned

HSV values stored externally in binary file

When testing I use the same test images for simple model

All simple tests use 8 bins in comparison histograms

Saved data does not need to be updated when changing between methods (method selected only needs changed)

***SAMPLE TABLE OF CONTENTS***

**Title Page**

**Into**

**GUI overview**

**Histograms**

**Correlation**

Simple Model

Bin optimization (precision and time)

Precision-Recall

Summary

**Chi-square**

Simple Model

Bin optimization (precision and time)

Precision-Recall

Summary

**Intersection**

Simple Model

Bin optimization (precision and time)

Precision-Recall

Summary

**Bhattacharyya**

Simple Model

Bin optimization (precision and time)

Precision-Recall

Summary

**Overall results**

**Database read/write results.**

Methods tried (CSV, .txt, SER, bin)

Results (time)

**Error handling**

**Possible improvements**

**Conclusion**

**References**

**Intro**

**Some Goals**

Experiment using different methods of calculating similarity/dissimilarity between a user entered query image and a set of images in a database.

The methods must incorporate differences in colour, shape, and texture.

Update image database in under 20 seconds.

Image search in under 5 seconds.

Would be willing to compromise slightly if there was a significant boost to accuracy/precision.

**GUI Overview**

**Histograms**

The figure below shows how I calculate HSV histograms.

This function takes as input a 3D array of float values representing an image in the HSV (hue, saturation, value) colour space, and an integer representing the number of bins to be used for each colour channel. The function returns a 3D array of float values representing the normalized histogram of the image in the HSV colour space.

It first creates an empty 3D array with dimensions equal to the number of colour channel bins specified. The step size for each bin is then calculated based on the range of values that can be represented in each colour channel. It then iterates through the input image, determining which bin each pixel belongs to, based on its HSV values. It increases the count in the histogram's corresponding bin. Finally, the histogram is normalised by dividing the count in each bin by the total number of pixels in the input image. The output is then the normalised histogram.

Text

Description automatically generated

This function is called in “compareHist.java” and “saveData.java”.

In “compareHist.java”, the histogram function is used for calculating a histogram from the query image.

And in “saveData.java” it is used to calculate the HSV histogram data for all of the images in the gallery so that it can be saved externally. This data will then later be read back in on future runs of the application and the data will be sent to “comapreHist.java” along with the query image mentioned above to calculate the similarity scores. Below shows how it is used in “saveData.java”.

Text

Description automatically generated

Text

Description automatically generatedBefore this is called in “saveData.java” I must construct a 3D HSV array, and it is this matrix that I send to the histogram function.

Text, letter

Description automatically generatedBelow shows the “RGBtoHSV” function that is used in the above function. I use HSV because it returns better results…

**Comparison Methods Selected**

* Correlation
* Chi-Square
* Intersection
* Bhattacharyya

All of the correlation, chi-square, intersection, and Bhattacharyya distance are distance metrics that can be used to compare image features.

Among these distance metrics, correlation and intersection are mainly used for shape features, while chi-square and Bhattacharyya distance can be used for both colour and texture features.

The chi-square distance measures the difference between two histograms, which can be used to compare colour distributions. It is commonly used in object recognition and retrieval applications.

The Bhattacharyya distance is a statistical measure that can be used to compare probability distributions, and it is commonly used to compare image colour and texture features.

Overall, using these distance metrics in this Java-based image search engine application will help perform efficient and accurate image analysis and comparison, ultimately resulting in better image retrieval and recognition capabilities.

\*\*include reference

**Text

Description automatically generatedCorrelation**

A picture containing text, clock, watch

Description automatically generatedwhere

*H1* and *H2* are the histograms being compared.

*N* is the total number of histogram bins.

For this method the correlation coefficient is a measure of how similar the two images are. The resulting correlation coefficient ranges from -1 to 1, where values closer to 1 indicate a better match between the template and the search image.

To get an optimal correlation model we want to determine what number of bins to use in our histograms when the images are compared to each other. We also want to record the time these processes take, because it is most desirable for this process to be as efficient as possible and not to sacrifice significant time for a marginal increase in model precision.

The code listing below shows how I have implemented this function onto my application.

Text

Description automatically generated

I will call this calculation method with every image in the database. I send the query image (“histBase”) and the current test image (“histTest”) as parameters so the newly calculated number can be returned back to the “compareHist.run” function where it is processed.

Below you can see the method being called and the result being stored in the double variable “baseTest”.

**Sample Results**

This section will demonstrate how this method works and how the comparison scores represent similar images. My application will return the top 15 most similar images to the GUI, based on the query image selected by the user.

8 bins used in this example

Query Image: *Crowds001.jpg*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0.84262344 | 0.824987434 | 0.81808332 | 0.801316703 | 0.77886204 | 0.764766727 |
| 0.736505927 | 0.719128692 | 0.688752314 | 0.672866897 | 0.671596902 | 0.65792797 |
| 0.656194644 | 0.654065127 | 0.638439513 | 0.631933075 | 0.628635661 | 0.598085047 |
| 0.587282513 | 0.58497173 | 0.570186753 | 0.566773157 | 0.565083048 | 0.540236183 |
| 0.526792272 | 0.521123343 | 0.502202935 | 0.476558587 | 0.476233572 | 0.472733289 |
| 0.471328041 | 0.46329864 | 0.458641672 | 0.45172543 | 0.450857056 | 0.449302965 |
| 0.449288812 | 0.448641871 | 0.446847079 | 0.438695024 | 0.423845276 | 0.402824346 |
| 0.392370555 | 0.382748833 | 0.381907916 | 0.380610703 | 0.374010195 | 0.365457501 |
| 0.362828153 | 0.35069428 | 0.342188379 | 0.337385222 | 0.337055568 | 0.330638523 |
| 0.329698592 | 0.32122675 | 0.318685518 | 0.317421815 | 0.315458732 | 0.315239366 |
| 0.309251386 | 0.308938355 | 0.308116762 | 0.302753944 | 0.301709561 | 0.300503704 |
| 0.289443456 | 0.288071181 | 0.286402738 | 0.283088251 | 0.280907469 | 0.269881831 |
| 0.267624255 | 0.266247765 | 0.265313802 | 0.264050294 | 0.264031792 | 0.255622495 |
| 0.249599115 | 0.248926477 | 0.24798834 | 0.246663108 | 0.242610199 | 0.237754996 |
| 0.231667315 | 0.228594344 | 0.228295256 | 0.22546458 | 0.222793611 | 0.222457817 |
| 0.221565658 | 0.220433594 | 0.210681811 | 0.209172044 | 0.208019091 | 0.206556288 |
| 0.199620017 | 0.199066401 | 0.198993215 | 0.192004676 | 0.187930091 | 0.186774765 |
| 0.18496253 | 0.183025781 | 0.182839281 | 0.181101801 | 0.179172399 | 0.175484334 |
| 0.174197568 | 0.171743801 | 0.170250849 | 0.168969315 | 0.1658904 | 0.163341378 |
| 0.162540758 | 0.157463242 | 0.154804362 | 0.15405846 | 0.153280946 | 0.152907262 |
| 0.151123101 | 0.149556672 | 0.149401706 | 0.14840628 | 0.147234933 | 0.145445699 |
| 0.145339196 | 0.145166165 | 0.143332828 | 0.141796777 | 0.140637069 | 0.140476242 |
| 0.140037075 | 0.14000401 | 0.139949272 | 0.13958114 | 0.137702433 | 0.137261534 |
| 0.136431326 | 0.135388863 | 0.13423435 | 0.131156478 | 0.130289927 | 0.129886344 |
| 0.127251238 | 0.126665459 | 0.12633979 | 0.126263799 | 0.124933834 | 0.12447742 |
| 0.123768074 | 0.123523653 | 0.12254895 | 0.119636786 | 0.119513821 | 0.119224944 |
| 0.119188814 | 0.116712533 | 0.116709551 | 0.116127171 | 0.115875846 | 0.11532518 |
| 0.114747784 | 0.114639722 | 0.114110961 | 0.11107077 | 0.110073825 | 0.108723136 |
| 0.10699626 | 0.105742922 | 0.104897557 | 0.104765208 | 0.102585779 | 0.101378501 |
| 0.100159244 | 0.098406057 | 0.09830978 | 0.09632434 | 0.096015695 | 0.095703153 |
| 0.095628246 | 0.095202184 | 0.093263907 | 0.092493167 | 0.091849458 | 0.091775367 |
| 0.091520235 | 0.091475985 | 0.088055172 | 0.08693757 | 0.084885737 | 0.083861223 |
| 0.082152887 | 0.080396442 | 0.080188451 | 0.07931637 | 0.078924189 | 0.07860063 |
| 0.078550155 | 0.075311426 | 0.074814067 | 0.072878409 | 0.070981542 | 0.070981136 |
| 0.068713817 | 0.068693565 | 0.067904053 | 0.067104779 | 0.066898646 | 0.066489554 |
| 0.0642638 | 0.06340873 | 0.062109305 | 0.059741183 | 0.05649781 | 0.056039862 |
| 0.055958206 | 0.055205092 | 0.054774101 | 0.054179942 | 0.053676489 | 0.051312904 |
| 0.051143917 | 0.050553687 | 0.050144938 | 0.050073118 | 0.048164584 | 0.04773922 |
| 0.047369396 | 0.047173639 | 0.045783837 | 0.044684247 | 0.044248082 | 0.044218808 |
| 0.043902328 | 0.043390092 | 0.043139348 | 0.042875828 | 0.0424868 | 0.042144341 |
| 0.041898207 | 0.04070462 | 0.040139586 | 0.039255739 | 0.038519027 | 0.038502692 |
| 0.037504892 | 0.03741361 | 0.037266846 | 0.036652401 | 0.035968057 | 0.035887568 |
| 0.035205474 | 0.034685187 | 0.034158993 | 0.0336975 | 0.033518572 | 0.033433097 |
| 0.033409909 | 0.033326976 | 0.03242287 | 0.032189744 | 0.032170598 | 0.032105934 |
| 0.030667795 | 0.030422539 | 0.029737893 | 0.028702833 | 0.028597086 | 0.028595995 |
| 0.028575037 | 0.028078629 | 0.027252646 | 0.026735551 | 0.02566026 | 0.025198552 |
| 0.024652512 | 0.023797735 | 0.02284173 | 0.022343716 | 0.021610981 | 0.020506262 |
| 0.019808231 | 0.019695633 | 0.019343366 | 0.017696677 | 0.017189186 | 0.016679305 |
| 0.016385607 | 0.01631217 | 0.01601723 | 0.015656283 | 0.014591648 | 0.013717717 |
| 0.013205954 | 0.011280957 | 0.011013653 | 0.01043529 | 0.010431729 | 0.009815149 |
| 0.009468652 | 0.009221347 | 0.009038603 | 0.008645476 | 0.008478442 | 0.00765197 |
| 0.006446356 | 0.001659077 | 0.001608657 | 0.00136622 | 0.001337174 | 0.001235642 |
| -0.000933795 | -0.001836113 | -0.001913736 | -0.002235978 | -0.002741722 | -0.003199447 |
| -0.003261053 | -0.004077037 | -0.005852751 | -0.005881474 | -0.007271949 | -0.008325626 |
| -0.008536279 | -0.009735119 | -0.010679274 | -0.011420862 | -0.014354543 | -0.017020388 |
| -0.019184284 | -0.02038066 | -0.020798229 | -0.023327716 | -0.023475476 | -0.026053736 |
| -0.03029712 | -0.030748769 | -0.031935674 | -0.032337354 | -0.032792033 | -0.033142769 |
| -0.034440389 | -0.034771998 | -0.040585627 |

This table above shows all the scores calculated for every comparison image in the available database of images, sorted from highest to lowest scoring. From this you can see that the function is working as all the scores fit the range from -1 to 1. Moreover, the resulting images have a precision of 100%.

A screenshot of a video game

Description automatically generated with medium confidenceThe image below shows the output from this. The 15 images displayed are represented in the table above are highlight in green.

**(*DISPLAY WORST PERFORMING IMAGES PERHAPS)***

**Simple Model**

I will use a starting point of bins = 8 for this baseline model. I will test 8 images and find the precision of the results and calculate an average time taken.

|  |  |  |
| --- | --- | --- |
| **Query Image** | **Results** | **Time (s)** |
| Crowds001.jpg | 100% | 2.615224 |
| Crowds10.jpg | 100% | 2.463153 |
| F1-Cars001.jpg | 7% | 2.394913 |
| F1-Cars10.jpg | 40% | 2.285437 |
| Horses001.jpg | 67% | 2.302323 |
| Horses10.jpg | 53% | 2.295571 |
| Landscapes001.jpg | 80% | 2.267539 |
| Landscapes10.jpg | 20% | 2.263979 |

|  |  |
| --- | --- |
| **Precision (%)** | **Mean Time (s)** |
| 58 | 2.361017363 |

**Precision-Recall Curve**

I will now produce a Precision-Recall Curve for each of these images, I will calculate the precision from the images over having 5, 10, 15, 20, 25 and 30 images returned. This will give a greater idea of the model’s performance and how its quality of results might drop off.

This gives a good base as to who the model may perform once the bins have been optimised…

**Bin Optimisation**

I will be returning 15 images.. based on results from initial precision-recall curve

Try intervals of 5. Until processing time becomes to great.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 5 | 49.17 | 0.775169038 |
| 10 | 51.00 | 4.576087713 |
| 15 | 49.17 | 14.30001306 |

This graph shows that when I increase the number of bins the processing time increases exponentially. I stopped this test at 15 bins as any processing time over that would clearly be undesirable for this application. Initially from this, the precision does not increase significantly with the bins, so it would be now checking each individual bins between 5 and 10 because a processing time of >4.58 would not be desirable. Especially considering that based off this graph the precision may not even increase.

Based on the results above we will now focus in and test each bin around the best performing interval. In this case this will be bins 5,6,7,8, and 9. This is because in the graph above. It appears that going up to bins around 15 doesn’t offer a boost to precision and the processing time is much too high. Moreover, bins <5 would offer very poor performance despite the low processing time.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 5 | 49.17 | 0.775169 |
| 6 | 53.33 | 1.160116 |
| 7 | 56.67 | 1.795463 |
| 8 | 57.5 | 2.514298 |
| 9 | 54.17 | 3.52963 |
| 10 | 51.00 | 4.576088 |
| 11 | 59.17 | 5.819499 |

This proved to be a good range of bins to check as it tests the upper limits of the time constraint (5 seconds). From this I can conclude that 8 bins would the best to use going forward as it offers the highest precision (57.5%), under 5 seconds.

Using 11 bins was a possibility but there’s a negative trade-off between a small increase in precision and relatively large increase in processing time.

**Full Precision-Recall Results**

Using 8 bins.

A picture containing text, watch, clock

Description automatically generated**Chi-Square**

*where*

*H1* and *H2* are the histograms being compared.

The distance measure is based on the idea of finding the minimum of the frequency values of each bin (or category) between the two histograms. The equation sums the minimum value of each bin across all the bins in the histogram, and the resulting value represents the dissimilarity between the two histograms.

The smaller the value of d(H1, H2), the more similar the histograms are, and the larger the value of d(H1, H2), the more dissimilar they are.

To get an optimal correlation model we want to determine what number of bins to use in our histograms when the images are compared to each other. We also want to record the time these processes take, because it is most desirable for this process to be as efficient as possible and not to sacrifice significant time for a marginal increase in model precision.

Text

Description automatically generatedThe code listing below shows how I have implemented this function onto my application.

I will call this calculation method with every image in the database. I send the query image (“histBase”) and the current test image (“histTest”) as parameters so the newly calculated number can be returned back to the “compareHist.run” function where it is processed.

Below you can see the method being called and the result being stored in the double variable “baseTest”.

**Sample Results**

This section will demonstrate how this method works and how the comparison scores represent similar images. My application will return the top 15 most similar images to the GUI, based on the query image selected by the user.

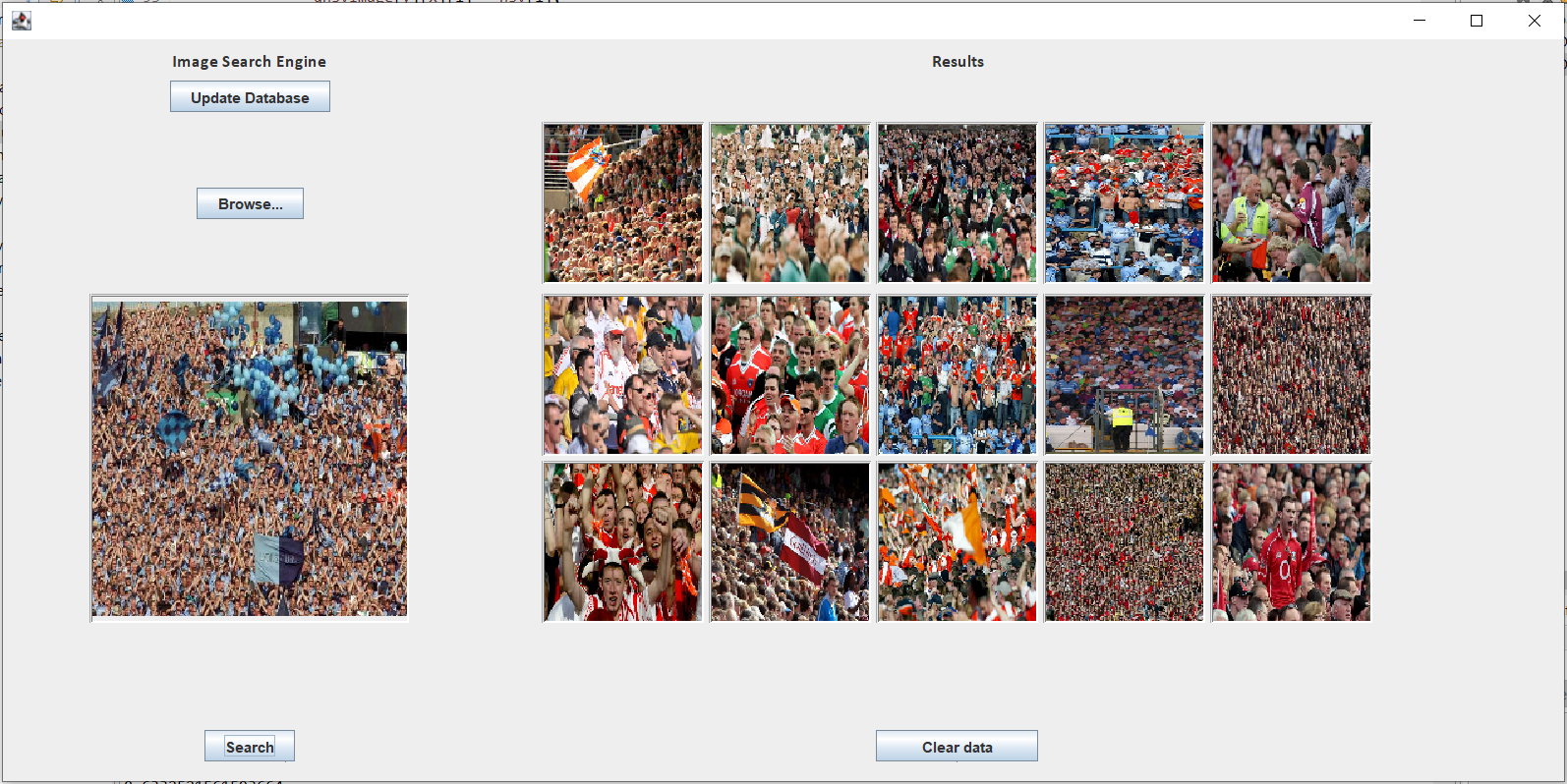
8 bins used in this example

Query Image: *Crowds001.jpg*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0.147198554 | 0.148594692 | 0.154034696 | 0.15676748 | 0.156990004 | 0.160475193 |
| 0.163228425 | 0.170389943 | 0.173957717 | 0.179043956 | 0.183769368 | 0.18728503 |
| 0.187364101 | 0.189503343 | 0.191980469 | 0.192485391 | 0.192964541 | 0.193424767 |
| 0.198962218 | 0.200548438 | 0.201671132 | 0.202986987 | 0.205520833 | 0.205894157 |
| 0.206728202 | 0.209259672 | 0.209501844 | 0.210236515 | 0.220428036 | 0.221153977 |
| 0.227993504 | 0.229566833 | 0.235518263 | 0.238827594 | 0.240618824 | 0.241889854 |
| 0.244555775 | 0.246915922 | 0.248578455 | 0.251529748 | 0.251601145 | 0.251691365 |
| 0.252254805 | 0.25576021 | 0.256159329 | 0.256890262 | 0.258346016 | 0.260196771 |
| 0.261466089 | 0.264023391 | 0.269366621 | 0.269379682 | 0.272756656 | 0.277390068 |
| 0.27882715 | 0.27901263 | 0.279944276 | 0.28171997 | 0.285461433 | 0.286976624 |
| 0.287083259 | 0.290863107 | 0.290945523 | 0.292846528 | 0.292868846 | 0.29373474 |
| 0.29477718 | 0.296741335 | 0.29987821 | 0.300380351 | 0.30169585 | 0.301778753 |
| 0.304940278 | 0.306141995 | 0.306143778 | 0.306222255 | 0.311621994 | 0.311966489 |
| 0.312694189 | 0.315976434 | 0.316862236 | 0.318233521 | 0.318957749 | 0.319008459 |
| 0.319187494 | 0.319334276 | 0.321783104 | 0.323335384 | 0.323535688 | 0.323976261 |
| 0.324456793 | 0.324834668 | 0.327430047 | 0.327824992 | 0.329190465 | 0.329757967 |
| 0.32993849 | 0.330101342 | 0.33086527 | 0.334073989 | 0.334696681 | 0.339299653 |
| 0.340177426 | 0.340393811 | 0.341662444 | 0.342034363 | 0.34240059 | 0.343054236 |
| 0.343186517 | 0.343478596 | 0.343652331 | 0.344731832 | 0.345486604 | 0.346827513 |
| 0.347088125 | 0.349099368 | 0.349425668 | 0.350468191 | 0.350894413 | 0.351407616 |
| 0.353622295 | 0.354616093 | 0.355412435 | 0.355974436 | 0.35707384 | 0.357793465 |
| 0.359227887 | 0.359496024 | 0.360818397 | 0.363151829 | 0.364007433 | 0.365356194 |
| 0.366522637 | 0.369046086 | 0.36968413 | 0.370525554 | 0.370895791 | 0.371067711 |
| 0.371203567 | 0.372569628 | 0.373951387 | 0.373996789 | 0.374436294 | 0.374567518 |
| 0.376094224 | 0.376900592 | 0.376912862 | 0.377646418 | 0.377736449 | 0.378456369 |
| 0.378544079 | 0.378578512 | 0.380432575 | 0.380685402 | 0.381564052 | 0.383005798 |
| 0.386281174 | 0.386557058 | 0.386612682 | 0.391252474 | 0.391898548 | 0.392305591 |
| 0.392453865 | 0.392941526 | 0.393179178 | 0.394860585 | 0.395384654 | 0.396196145 |
| 0.397063315 | 0.39752171 | 0.401195352 | 0.402726154 | 0.403142986 | 0.403156236 |
| 0.405195943 | 0.40529929 | 0.405316478 | 0.408809945 | 0.409982993 | 0.410916171 |
| 0.411680801 | 0.411730744 | 0.412314148 | 0.413138678 | 0.413148394 | 0.413672697 |
| 0.413717548 | 0.414128227 | 0.414311554 | 0.416209874 | 0.41715881 | 0.419171853 |
| 0.419382611 | 0.421413074 | 0.421767184 | 0.422159688 | 0.422444845 | 0.423755742 |
| 0.42533172 | 0.425671694 | 0.42690941 | 0.427321835 | 0.428445815 | 0.428774565 |
| 0.42888076 | 0.429408494 | 0.429580406 | 0.431026086 | 0.431354033 | 0.432242076 |
| 0.432326081 | 0.433014397 | 0.433662377 | 0.4342688 | 0.434275933 | 0.435677163 |
| 0.43603277 | 0.436331077 | 0.436380179 | 0.43673743 | 0.436807902 | 0.437329353 |
| 0.438035346 | 0.438398396 | 0.440347528 | 0.441816842 | 0.442383728 | 0.444798633 |
| 0.44486778 | 0.444880618 | 0.445024844 | 0.445657721 | 0.446254226 | 0.446900975 |
| 0.44746611 | 0.45043229 | 0.45052963 | 0.451517721 | 0.452278016 | 0.45266071 |
| 0.453124254 | 0.453329622 | 0.453676652 | 0.455250072 | 0.455653509 | 0.457157369 |
| 0.458005923 | 0.458164735 | 0.458400293 | 0.45942914 | 0.459505484 | 0.459682439 |
| 0.460185558 | 0.462658359 | 0.462745539 | 0.463035897 | 0.463200882 | 0.463431616 |
| 0.46430294 | 0.464712649 | 0.465036878 | 0.466062245 | 0.466711904 | 0.4667731 |
| 0.467118 | 0.468159288 | 0.469738509 | 0.47254708 | 0.474920949 | 0.475445658 |
| 0.475633674 | 0.47751989 | 0.477753676 | 0.477900742 | 0.477969112 | 0.478531231 |
| 0.478543925 | 0.478570914 | 0.478782753 | 0.479872407 | 0.480372822 | 0.481453714 |
| 0.48148805 | 0.48315875 | 0.48382811 | 0.48886258 | 0.488914963 | 0.488978048 |
| 0.489165647 | 0.489201363 | 0.489704693 | 0.491250899 | 0.491911881 | 0.492545143 |
| 0.494670697 | 0.495672017 | 0.496006515 | 0.496951566 | 0.498562612 | 0.498812269 |
| 0.498900503 | 0.498900719 | 0.499340836 | 0.501094223 | 0.503501856 | 0.504779864 |
| 0.504812668 | 0.505950194 | 0.506233791 | 0.506480916 | 0.507953826 | 0.508450214 |
| 0.508474754 | 0.509531263 | 0.509567685 | 0.510405248 | 0.512145268 | 0.51286646 |
| 0.514457898 | 0.51675439 | 0.520174843 | 0.521004758 | 0.526480502 | 0.527517748 |
| 0.528798537 | 0.531535443 | 0.536475032 | 0.536563908 | 0.537189437 | 0.539274281 |
| 0.540655308 | 0.541663914 | 0.541850002 | 0.546489787 | 0.558288064 | 0.560897167 |
| 0.562934947 | 0.564341754 | 0.565780552 | 0.567216507 | 0.576501257 | 0.576562763 |
| 0.579712104 | 0.597204841 | 0.633252156 |

This table above shows all the scores calculated for every comparison image in the available database of images, sorted from lowest to highest scoring. From this you can see that the function is working as all the scores fit the range from 0 to 1. Moreover, the resulting images have a precision of 100%.

The image below shows the output from this. The 15 images displayed are represented in the table above are highlight in green.

**(*DISPLAY WORST PERFORMING IMAGES PERHAPS)***

**Simple Model**

I will use a starting point of bins = 8 for this baseline model. I will test 8 images and find the precision of the results and calculate an average time taken.

|  |  |  |
| --- | --- | --- |
| **Query Image** | **Results** | **Time (s)** |
| Crowds001.jpg | 100% | 3.9969346 |
| Crowds10.jpg | 100% | 3.708615 |
| F1-Cars001.jpg | 80% | 3.7084899 |
| F1-Cars10.jpg | 47% | 3.4998552 |
| Horses001.jpg | 73% | 3.5586724 |
| Horses10.jpg | 40% | 3.705902 |
| Landscapes001.jpg | 80% | 3.7998657 |
| Landscapes10.jpg | 27% | 3.509239 |

|  |  |
| --- | --- |
| **Precision (%)** | **Mean Time (s)** |
| 68.38 | 3.685946725 |

**Precision-Recall Curve**

I will now produce a Precision-Recall Curve for each of these images, I will calculate the precision from the images over having 5, 10, 15, 20, 25 and 30 images returned. This will give a greater idea of the model’s performance and how its quality of results might drop off.

This gives a good base as to who the model may perform once the bins have been optimised…

**Bin Optimisation**

I will be returning 15 images.. based on results from initial precision-recall curve

Try intervals of 5. Until processing time becomes to great.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 5 | 49.17 | 2.26 |
| 10 | 50 | 5.72 |
| 15 | 45 | 15.90 |

This graph shows that when I increase the number of bins from 5 to 10 the precision increases slightly. Bins greater than 10 should be discarded as the processing time would be too highest and based on this test would not offer an increase in precision. I stopped this test at 15 bins as any processing time over that would clearly be undesirable for this application.

Based on the results above we will now focus in and additionally test each bin around the best performing interval. In this case this will be bins 6,7,8, and 9. It appears that going above 10 bins doesn’t offer a boost to precision and the processing time is too high. Moreover, bins <5 would offer very poor performance despite the low processing time.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 4 | 50.83 | 1.804158 |
| 5 | 49.17 | 2.258742 |
| 6 | 54.17 | 2.40748 |
| 7 | 57.5 | 3.070594 |
| 8 | 59.17 | 3.569663 |
| 9 | 55 | 4.955153 |

This proved to be a good range of bins to check as it tests the upper limits of the time constraint (5 seconds). From this I can conclude that 8 bins would the best to use going forward as it offers the highest precision (59.17%), under 5 seconds.

**Full Precision-Recall Results**

Using 8 bins

A picture containing text

Description automatically generated**Intersection**

*where*

*H1* and *H2* are the histograms being compared.

The equation works by comparing the values in each bin or category of the two histograms and selecting the minimum value between them. For each bin "I", the minimum value between H1(I) and H2(I) is selected and summed across all the bins in the histograms. The resulting value represents the distance between the two histograms.

Intuitively, this distance measure captures the idea of finding the "intersection" between the two histograms, by comparing the minimum frequency of each bin. If the two histograms are identical, the distance will be zero, as each bin in both histograms will have the same frequency. On the other hand, if the two histograms are very dissimilar, the distance will be large, as there will be very little overlap/commonality between the bins of the two histograms.

To get an optimal correlation model we want to determine what number of bins to use in our histograms when the images are compared to each other. We also want to record the time these processes take, because it is most desirable for this process to be as efficient as possible and not to sacrifice significant time for a marginal increase in model precision.

Graphical user interface, text

Description automatically generatedThe code listing below shows how I have implemented this function onto my application.

I will call this calculation method with every image in the database. I send the query image (“histBase”) and the current test image (“histTest”) as parameters so the newly calculated number can be returned back to the “compareHist.run” function where it is processed.

Below you can see the method being called and the result being stored in the double variable “baseTest”.



**Sample Results**

This section will demonstrate how this method works and how the comparison scores represent similar images. My application will return the top 15 most similar images to the GUI, based on the query image selected by the user.

8 bins used in this example

Query Image: *Crowds001.jpg*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0.010299683 | 0.023466746 | 0.028322855 | 0.02897644 | 0.029860179 | 0.031131744 |
| 0.031205495 | 0.033870697 | 0.033912659 | 0.036047617 | 0.038022359 | 0.039609273 |
| 0.040112813 | 0.040566762 | 0.041027069 | 0.04162852 | 0.042077382 | 0.043102264 |
| 0.043543497 | 0.043621063 | 0.043641408 | 0.045459747 | 0.046005249 | 0.046731313 |
| 0.047590891 | 0.04909261 | 0.049922943 | 0.049978892 | 0.050331116 | 0.050412496 |
| 0.051836649 | 0.052346547 | 0.052436828 | 0.052687327 | 0.054059347 | 0.0542895 |
| 0.054506938 | 0.054601034 | 0.05480957 | 0.054948171 | 0.055019378 | 0.055233001 |
| 0.055437724 | 0.055768331 | 0.055781046 | 0.056056976 | 0.056191762 | 0.056435903 |
| 0.056512197 | 0.056625366 | 0.056714376 | 0.056971232 | 0.057239532 | 0.057657877 |
| 0.058231354 | 0.058795929 | 0.059131622 | 0.059505462 | 0.060001373 | 0.060198466 |
| 0.060451508 | 0.060559591 | 0.061439514 | 0.061481476 | 0.06161499 | 0.062104543 |
| 0.062526703 | 0.062689463 | 0.063284556 | 0.063423157 | 0.063721975 | 0.064423879 |
| 0.064636231 | 0.064856211 | 0.064879099 | 0.064966837 | 0.065724691 | 0.065865835 |
| 0.066037496 | 0.066799164 | 0.066936493 | 0.066958109 | 0.068042754 | 0.068180084 |
| 0.068401337 | 0.068843841 | 0.06887563 | 0.068976084 | 0.06935501 | 0.069653829 |
| 0.069746653 | 0.06982549 | 0.070134481 | 0.070217132 | 0.070435841 | 0.070486705 |
| 0.07070287 | 0.07089742 | 0.070983886 | 0.071153005 | 0.071371714 | 0.072448731 |
| 0.072731018 | 0.073066711 | 0.07314682 | 0.073397318 | 0.073654175 | 0.07423528 |
| 0.07452774 | 0.074709574 | 0.075088501 | 0.075215658 | 0.075228373 | 0.075578054 |
| 0.075785319 | 0.076110839 | 0.076148986 | 0.076501211 | 0.076615651 | 0.076978047 |
| 0.077419281 | 0.077925364 | 0.078062693 | 0.078235626 | 0.078479767 | 0.078564962 |
| 0.078699747 | 0.079938253 | 0.080111186 | 0.080214182 | 0.0802447 | 0.080266317 |
| 0.080403645 | 0.080457051 | 0.080557505 | 0.080640157 | 0.08070755 | 0.080949147 |
| 0.080979665 | 0.0811704 | 0.081563313 | 0.081573486 | 0.082027435 | 0.082191467 |
| 0.082688649 | 0.08275477 | 0.082883199 | 0.084273021 | 0.085268657 | 0.085489909 |
| 0.085534413 | 0.085586548 | 0.085840861 | 0.086125692 | 0.086716969 | 0.086744944 |
| 0.086772919 | 0.08690389 | 0.086907705 | 0.087284088 | 0.087861379 | 0.088064829 |
| 0.088134765 | 0.089071909 | 0.089731852 | 0.08986028 | 0.090601603 | 0.090602875 |
| 0.09082667 | 0.091147104 | 0.091546376 | 0.091810862 | 0.092351277 | 0.09250768 |
| 0.092894236 | 0.093199412 | 0.093420664 | 0.093611399 | 0.094445546 | 0.094484965 |
| 0.09464391 | 0.094969431 | 0.095637003 | 0.095938364 | 0.096539815 | 0.097722371 |
| 0.097886403 | 0.099203745 | 0.099259694 | 0.099268595 | 0.100280762 | 0.1003685 |
| 0.100397746 | 0.100963593 | 0.101006826 | 0.101062774 | 0.101107279 | 0.101648966 |
| 0.101815541 | 0.102390289 | 0.102750142 | 0.102884928 | 0.103696187 | 0.103713989 |
| 0.103900909 | 0.104230244 | 0.104911804 | 0.10543696 | 0.105806986 | 0.105809529 |
| 0.106380462 | 0.106433868 | 0.106685638 | 0.107116698 | 0.107643127 | 0.107856751 |
| 0.108690897 | 0.108720144 | 0.109279632 | 0.109348297 | 0.109450022 | 0.109545389 |
| 0.110123952 | 0.110459646 | 0.110984802 | 0.111003875 | 0.111846923 | 0.11251831 |
| 0.11267217 | 0.113623301 | 0.113883972 | 0.114058175 | 0.1142807 | 0.114354451 |
| 0.114496867 | 0.115328471 | 0.115918477 | 0.116841634 | 0.116970062 | 0.116984049 |
| 0.117158254 | 0.118770599 | 0.118905385 | 0.11900711 | 0.119116465 | 0.119326274 |
| 0.120855967 | 0.121081034 | 0.121720632 | 0.121772766 | 0.121836344 | 0.122450511 |
| 0.122624714 | 0.122714996 | 0.122891744 | 0.1236763 | 0.123685201 | 0.123797098 |
| 0.124365489 | 0.125334421 | 0.125765483 | 0.125862121 | 0.128223418 | 0.128369649 |
| 0.129936219 | 0.130208333 | 0.130554198 | 0.130869547 | 0.131444295 | 0.131896972 |
| 0.132125854 | 0.132441203 | 0.132471719 | 0.132663727 | 0.132704417 | 0.133363088 |
| 0.133926392 | 0.134848276 | 0.135735829 | 0.136330921 | 0.137914021 | 0.137989043 |
| 0.13831838 | 0.138483683 | 0.139575959 | 0.140280405 | 0.140892028 | 0.141058604 |
| 0.14136378 | 0.142042795 | 0.14206187 | 0.143361409 | 0.143992105 | 0.145366669 |
| 0.146737416 | 0.147335052 | 0.149004617 | 0.149426778 | 0.151124318 | 0.152222952 |
| 0.152286529 | 0.153355916 | 0.15380605 | 0.154257455 | 0.154490153 | 0.154588063 |
| 0.15618515 | 0.157449087 | 0.159510294 | 0.160049438 | 0.160162608 | 0.161191304 |
| 0.161674499 | 0.16676712 | 0.166983286 | 0.167794546 | 0.168028513 | 0.170899708 |
| 0.171440124 | 0.174892425 | 0.176076253 | 0.176120759 | 0.176222484 | 0.176315307 |
| 0.176320394 | 0.178910572 | 0.179880778 | 0.183583578 | 0.183715821 | 0.184140522 |
| 0.185376485 | 0.185646056 | 0.186153412 | 0.186553955 | 0.190457661 | 0.192638397 |
| 0.192774455 | 0.193851471 | 0.195364634 | 0.201718647 | 0.201817831 | 0.206745149 |
| 0.208287558 | 0.210079194 | 0.21096166 |

This table above shows all the scores calculated for every comparison image in the available database of images, sorted from lowest to highest scoring. From this you can see that the function is working as all the scores fit the range from 0 to 1, where the values closer to 0 are the most similar. Moreover, the resulting images have a precision of 100%.

The image below shows the output from this. The 15 images displayed are represented in the table above are highlight in green.

**(*DISPLAY WORST PERFORMING IMAGES PERHAPS)***

A screenshot of a video game

Description automatically generated with medium confidence**Simple Model**

I will use a starting point of bins = 8 for this baseline model. I will test 8 images and find the precision of the results and calculate an average time taken.

|  |  |  |
| --- | --- | --- |
| **Query Image** | **Results** | **Time (s)** |
| Crowds001.jpg | 100% | 2.5834942 |
| Crowds10.jpg | 100% | 2.3988715 |
| F1-Cars001.jpg | 7% | 2.3432522 |
| F1-Cars10.jpg | 60% | 2.4349141 |
| Horses001.jpg | 73% | 2.329884 |
| Horses10.jpg | 40% | 2.3156995 |
| Landscapes001.jpg | 60% | 2.3262559 |
| Landscapes10.jpg | 27% | 2.356708 |

|  |  |
| --- | --- |
| **Precision (%)** | **Mean Time (s)** |
| 58.38 | 2.386134925 |

**Precision-Recall Curve**

I will now produce a Precision-Recall Curve for each of these images, I will calculate the precision from the images over having 5, 10, 15, 20, 25 and 30 images returned. This will give a greater idea of the model’s performance and how its quality of results might drop off.

This gives a good base as to who the model may perform once the bins have been optimised…

**Bin Optimisation**

I will be returning 15 images.. based on results from initial precision-recall curve

Try intervals of 5. Until processing time becomes too great.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 5 | 48.33 | 0.77 |
| 10 | 51.67 | 4.40 |
| 15 | 50.83 | 14.49 |

This graph shows that when I increase the number of bins from 5 to 10 the precision and processing time increases. Based on this graph it seems that 11 bins may be able to stay below the 5 second processing but any more would be unusable. I stopped this test at 15 bins as any processing time over 14.49s would clearly be undesirable for this application.

Based on the results above we will now focus in and additionally test each bin around the best performing interval. In this case this will be bins 7,8,9, and 11. The above graph suggests bins less than 5 would offer very poor performance despite the low processing time.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 7 | 58.33 | 1.75 |
| 8 | 59.17 | 2.39 |
| 9 | 55.83 | 3.34 |
| 10 | 51.67 | 4.400278 |
| 11 | 58.33 | 5.971803 |

This proved to be a good range of bins to check as it tests the upper limits of the time constraint (5 seconds). From this I can conclude that 8 bins would the best to use going forward as it offers the highest precision (59.17%), under 5 seconds.

**Full Precision-Recall Results**

Using 8 bins.

A picture containing text, antenna

Description automatically generated**Bhattacharyya**

*where*

*H1* and *H2* are the histograms being compared.

*N* is the total number of histogram bins.

The Bhattacharyya distance is a statistical measure of the similarity between two probability distributions. It is calculated based on the overlap between the two distributions, and it considers both the mean and variance of the distributions.

In the above equation the numerator of the equation calculates the covariance between the two histograms, and the denominator of the equation is a normalization factor that scales the covariance value by the product of the means and the number of bins in the histograms.

The resulting Bhattacharyya distance is a value between 0 and 1, where 0 indicates a perfect match between the two histograms, and 1 indicates no overlap or similarity between the two histograms.

To get an optimal correlation model we want to determine what number of bins to use in our histograms when the images are compared to each other. We also want to record the time these processes take, because it is most desirable for this process to be as efficient as possible and not to sacrifice significant time for a marginal increase in model precision.

The code listing below shows how I have implemented this function onto my application.Text

Description automatically generated

I will call this calculation method with every image in the database. I send the query image (“histBase”) and the current test image (“histTest”) as parameters so the newly calculated number can be returned back to the “compareHist.run” function where it is processed.

Below you can see the method being called and the result being stored in the double variable “baseTest”.

**Sample Results**

This section will demonstrate how this method works and how the comparison scores represent similar images. My application will return the top 15 most similar images to the GUI, based on the query image selected by the user.

8 bins used in this example

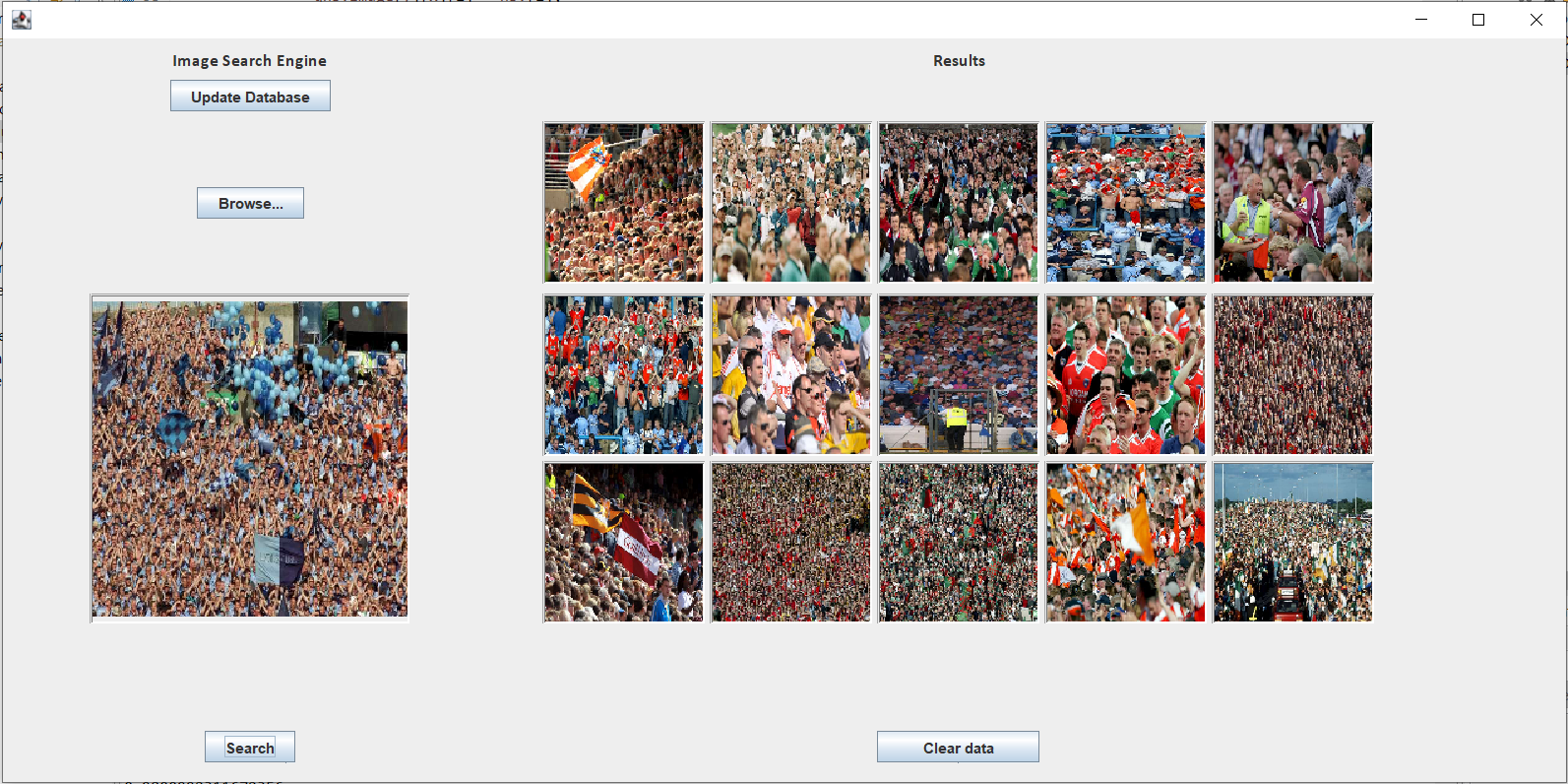
Query Image: *Crowds001.jpg*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0.844687746 | 0.84486907 | 0.845155693 | 0.84540345 | 0.846937671 | 0.848905089 |
| 0.849337502 | 0.849969806 | 0.850310239 | 0.850454253 | 0.8517765 | 0.852494668 |
| 0.853439933 | 0.85399739 | 0.854391166 | 0.85446577 | 0.8546489 | 0.854931081 |
| 0.855146408 | 0.85543718 | 0.855712274 | 0.855982623 | 0.856093682 | 0.856732133 |
| 0.857033735 | 0.857372816 | 0.858212723 | 0.858788834 | 0.859986795 | 0.860485726 |
| 0.862474506 | 0.862893948 | 0.863399949 | 0.863545914 | 0.86362269 | 0.863685095 |
| 0.863704548 | 0.864597075 | 0.86518504 | 0.865689182 | 0.866598701 | 0.866706406 |
| 0.867068755 | 0.867215303 | 0.867724405 | 0.867879416 | 0.868441626 | 0.86899869 |
| 0.870005512 | 0.870491189 | 0.870844059 | 0.871227552 | 0.871262874 | 0.87155719 |
| 0.871603638 | 0.871905264 | 0.872391617 | 0.872452531 | 0.873039144 | 0.873439944 |
| 0.874071015 | 0.874526663 | 0.874529664 | 0.875149208 | 0.875318145 | 0.875963621 |
| 0.87624388 | 0.877743197 | 0.877995122 | 0.878255189 | 0.878439902 | 0.878567221 |
| 0.878603749 | 0.87866497 | 0.879103086 | 0.879807812 | 0.881455927 | 0.88170915 |
| 0.882038697 | 0.882450067 | 0.882974757 | 0.883132105 | 0.883341411 | 0.883368626 |
| 0.883459344 | 0.883543372 | 0.883638147 | 0.883771209 | 0.8839412 | 0.883947284 |
| 0.884249179 | 0.884326602 | 0.884416101 | 0.88465453 | 0.884680495 | 0.884724119 |
| 0.885105254 | 0.886155362 | 0.886331691 | 0.886546769 | 0.886814595 | 0.886914658 |
| 0.88757254 | 0.887701876 | 0.887931217 | 0.887997256 | 0.888028878 | 0.888295905 |
| 0.888529509 | 0.889077143 | 0.889382078 | 0.889667594 | 0.890091961 | 0.890547565 |
| 0.890571423 | 0.89071067 | 0.89074222 | 0.890756459 | 0.89119813 | 0.891586454 |
| 0.891675874 | 0.891747578 | 0.891909783 | 0.891963531 | 0.892364526 | 0.892759656 |
| 0.892763604 | 0.892954356 | 0.893364377 | 0.893402232 | 0.893590366 | 0.893665348 |
| 0.893810019 | 0.894256211 | 0.894287483 | 0.894833686 | 0.895313273 | 0.89547253 |
| 0.895486867 | 0.895844715 | 0.895939 | 0.895983966 | 0.896025321 | 0.896177618 |
| 0.896310462 | 0.896671989 | 0.896673665 | 0.896858399 | 0.897132254 | 0.897409943 |
| 0.89789598 | 0.897899905 | 0.897928527 | 0.898902082 | 0.899513251 | 0.899752249 |
| 0.899829125 | 0.899861939 | 0.900003554 | 0.900006376 | 0.900143575 | 0.90016131 |
| 0.900281795 | 0.900416062 | 0.900596367 | 0.900871465 | 0.901066844 | 0.901629561 |
| 0.901877369 | 0.902340069 | 0.902657055 | 0.902833685 | 0.903238494 | 0.903287144 |
| 0.903418751 | 0.904146749 | 0.90420107 | 0.904727017 | 0.904895613 | 0.905022572 |
| 0.906057213 | 0.906083 | 0.906120597 | 0.906131074 | 0.906601853 | 0.907305995 |
| 0.907455292 | 0.907848668 | 0.907884864 | 0.907956415 | 0.908102246 | 0.908379985 |
| 0.90850407 | 0.908593927 | 0.90943463 | 0.909470678 | 0.909491898 | 0.909502715 |
| 0.910106929 | 0.910115664 | 0.910212764 | 0.910333753 | 0.910459179 | 0.910481885 |
| 0.910788503 | 0.910839802 | 0.910859815 | 0.91092727 | 0.91095673 | 0.911019199 |
| 0.911062521 | 0.911086105 | 0.911116741 | 0.911141858 | 0.911242176 | 0.911309232 |
| 0.91164368 | 0.911723306 | 0.911858109 | 0.912260154 | 0.913138814 | 0.913290513 |
| 0.913467652 | 0.913879909 | 0.913941072 | 0.914025273 | 0.91530748 | 0.915308986 |
| 0.915527087 | 0.915632037 | 0.915736437 | 0.91608296 | 0.916165527 | 0.916264146 |
| 0.916285825 | 0.916358291 | 0.916447486 | 0.916509019 | 0.916661451 | 0.917114111 |
| 0.917129997 | 0.917530099 | 0.917660487 | 0.918146987 | 0.918336515 | 0.918633324 |
| 0.918997203 | 0.919020783 | 0.919331068 | 0.919353828 | 0.919430997 | 0.919570118 |
| 0.919975514 | 0.920125091 | 0.920139587 | 0.920429377 | 0.920749288 | 0.920953061 |
| 0.921496837 | 0.921550861 | 0.922718344 | 0.922923259 | 0.923125376 | 0.923267626 |
| 0.923277028 | 0.923623599 | 0.923650781 | 0.923735396 | 0.923814381 | 0.924078687 |
| 0.924147492 | 0.924307339 | 0.924612308 | 0.925014967 | 0.925384159 | 0.925751233 |
| 0.925954825 | 0.926099694 | 0.926313118 | 0.926393128 | 0.926474989 | 0.926518451 |
| 0.926960985 | 0.927115199 | 0.927771371 | 0.928172396 | 0.928204123 | 0.928226878 |
| 0.928331353 | 0.928387482 | 0.928648684 | 0.928675219 | 0.928709971 | 0.92933007 |
| 0.929616116 | 0.930066848 | 0.930456578 | 0.930473612 | 0.930621696 | 0.930651408 |
| 0.930798849 | 0.931053286 | 0.931084595 | 0.931151376 | 0.931239494 | 0.931522679 |
| 0.932483843 | 0.932633086 | 0.93315663 | 0.933209617 | 0.93335945 | 0.933429265 |
| 0.933471363 | 0.933577098 | 0.934802725 | 0.936291778 | 0.936459635 | 0.93648351 |
| 0.93662399 | 0.937233112 | 0.937935231 | 0.938083305 | 0.93836208 | 0.938401293 |
| 0.939972831 | 0.940047795 | 0.943516382 | 0.943636617 | 0.943749016 | 0.944871275 |
| 0.94564848 | 0.946274883 | 0.947042886 | 0.948791439 | 0.950367479 | 0.952598906 |
| 0.952676156 | 0.952907732 | 0.953241656 | 0.956700668 | 0.958850743 | 0.959562391 |
| 0.962875106 | 0.965180724 | 0.980099231 |

This table above shows all the scores calculated for every comparison image in the available database of images, sorted from lowest to highest scoring. From this you can see that the function is working as all the scores fit the range from 0 to 1, where the values closer to 0 are the most similar. Moreover, the resulting images have a precision of 100%.

The image below shows the output from this. The 15 images displayed are represented in the table above are highlight in green.

**(*DISPLAY WORST PERFORMING IMAGES PERHAPS)***



**Simple Model**

I will use a starting point of bins = 8 for this baseline model. I will test 8 images and find the precision of the results and calculate an average time taken.

|  |  |  |
| --- | --- | --- |
| **Query Image** | **Results** | **Time (s)** |
| Crowds001.jpg | 100% | 4.0466897 |
| Crowds10.jpg | 100% | 6.378527 |
| F1-Cars001.jpg | 7% | 3.5315454 |
| F1-Cars10.jpg | 47% | 3.6238344 |
| Horses001.jpg | 60% | 3.6727772 |
| Horses10.jpg | 33% | 3.6193182 |
| Landscapes001.jpg | 73% | 3.4608025 |
| Landscapes10.jpg | 20% | 3.4783416 |

|  |  |
| --- | --- |
| **Precision (%)** | **Mean Time (s)** |
| 55 | 3.9764795 |

**Precision-Recall Curve**

I will now produce a Precision-Recall Curve for each of these images, I will calculate the precision from the images over having 5, 10, 15, 20, 25 and 30 images returned. This will give a greater idea of the model’s performance and how its quality of results might drop off.

This gives a good base as to who the model may perform once the bins have been optimised…

**Bin Optimisation**

I will be returning 15 images.. based on results from initial precision-recall curve

Try intervals of 5. Until processing time becomes too great.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 5 | 52.5 | 1.97 |
| 10 | 48.33 | 5.74 |
| 15 | 45 | 15.75 |

This graph differs from the initial bin optimisation for the other models as it appears that the precision decreases as the bin number is increased. Despite this processing time acts normally as it clearly increases.

Based on this graph it seems that any more than 10 bins would be unusable, as it would breach the 5 second processing limit. I stopped this test at 15 bins as any processing time over 15.75 would clearly be undesirable for this application.

Based on the results above we will now focus in and additionally test each bin around the best performing interval. In this case this will be bins 4,6,7,8, and 9.

|  |  |  |
| --- | --- | --- |
| **Bin Number** | **Average Results (%)** | **Average Time (s)** |
| 4 | 50.83 | 1.74 |
| 5 | 52.5 | 1.97 |
| 6 | 56.67 | 2.52 |
| 7 | 60.83 | 2.782253 |
| 8 | 56.67 | 3.672113 |
| 9 | 56.67 | 4.51194095 |

This proved to be a good range of bins to check as it tests the upper limits of the time constraint (5 seconds). From this I can conclude that 7 bins would the best to use going forward as it offers the highest precision (56.67%), under 5 seconds.

**Full Precision-Recall Results**

Using 7 bins.

**Overall Results**

**Simple Models**

The table below shows the results from a very basic initial test to help give a better understanding as to how each of the models may perform in comparison to each other. In this test 15 images are being returned and I am using the optimal bin number for each comparison method.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Query Image** | **Results** | **Time (s)** |
| Correlation | Crowds001.jpg | 15.00 | 2.596444 |
| (8 bins) | Crowds10.jpg | 15.00 | 2.929038 |
|  | F1-Cars001.jpg | 1.00 | 2.580701 |
|  | F1-Cars10.jpg | 6.00 | 2.614612 |
|  | Horses001.jpg | 10.00 | 2.276192 |
|  | Horses10.jpg | 8.00 | 2.401694 |
|  | Landscapes001.jpg | 11.00 | 2.29146 |
|  | Landscapes10.jpg | 3.00 | 2.424246 |
|  |  | **57.5** | **2.514298** |
|  |  |  |  |
| Chi-square | Crowds001.jpg | 15.00 | 3.841582 |
| (8 bins) | Crowds10.jpg | 15.00 | 3.59576 |
|  | F1-Cars001.jpg | 1.00 | 3.495053 |
|  | F1-Cars10.jpg | 7.00 | 3.556995 |
|  | Horses001.jpg | 11.00 | 3.524728 |
|  | Horses10.jpg | 6.00 | 3.427903 |
|  | Landscapes001.jpg | 12.00 | 3.59234 |
|  | Landscapes10.jpg | 4.00 | 3.522942 |
|  |  | **59.17** | **3.569663** |
|  |  |  |  |
| Intersection | Crowds001.jpg | 15.00 | 2.409595 |
| (8 bins) | Crowds10.jpg | 15.00 | 2.377954 |
|  | F1-Cars001.jpg | 1.00 | 2.284409 |
|  | F1-Cars10.jpg | 9.00 | 2.29746 |
|  | Horses001.jpg | 11.00 | 2.408053 |
|  | Horses10.jpg | 7.00 | 2.479597 |
|  | Landscapes001.jpg | 9.00 | 2.467109 |
|  | Landscapes10.jpg | 4.00 | 2.391278 |
|  |  | **59.17** | **2.389432** |
|  |  |  |  |
| Bhattacharyya | Crowds001.jpg | 14.00 | 2.827121 |
| (7 bins) | Crowds10.jpg | 15.00 | 2.967066 |
|  | F1-Cars001.jpg | 1.00 | 2.745372 |
|  | F1-Cars10.jpg | 8.00 | 2.797245 |
|  | Horses001.jpg | 11.00 | 2.759143 |
|  | Horses10.jpg | 8.00 | 2.734024 |
|  | Landscapes001.jpg | 12.00 | 2.723621 |
|  | Landscapes10.jpg | 4.00 | 2.704434 |
|  |  | **60.83** | **2.782253** |

The best model will have high precision and low time.

As all of these models have optimal bins already chosen, time should not be a major issue and it is primarily the best precision that matters the most.

It is clear to see that Correlation is the worst performing model as its precision is the lowest at only 57.5% despite the fact its processing time is relatively low at just 2.51 seconds.

Chi-Square and Intersection have both performed the same in terms of precision in this test, recording 59.17%. Despite this Intersection would still be preferable because it has a lower processing time. 3.569663 seconds compared to 2.389432 seconds, means that Intersection is 1.180231 seconds faster than Chi-Square.

Bhattacharya initially appears to be the best performing model as it has the highest precision (60.83%). Although it does have the second highest processing time behind Chi-Square, this should not be an issue as this time is acceptable.

|  |  |
| --- | --- |
| **Method** | **Precision (%) / time (s) (2dp)** |
| Correlation | 22.87 |
| Chi-Square | 16.58 |
| Intersection | 24.76 |
| Bhattacharyya | 21.86 |

If we look deeper into the efficiency of these models (as seen in the figure above), we can see that despite Bhattacharyya being the most precise, it has the second lowest efficiency according to this measure (behind Chi-Square). Moreover, it is also important to note that Intersection has the highest efficiency at “24.76” so it might perform better when we explore these methods and results further.

**Simple Precision-Recall Averages**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Num Bins** | **Correlation** | **Chi-Square** | **Intersection** | **Bhattacharyya** |
| **5** | 0.45 | 0.65 | 0.58 | 0.63 |
| **10** | 0.53 | 0.60 | 0.59 | 0.59 |
| **15** | 0.59 | 0.59 | 0.57 | 0.57 |
| **20** | 0.57 | 0.56 | 0.54 | 0.54 |
| **25** | 0.54 | 0.55 | 0.52 | 0.54 |
| **30** | 0.5 | 0.52 | 0.51 | 0.51 |

The data for this graph and table is taken from the average trend line from the basic Precision-Recall Curves mention previously in each of the methods sections.

These results are conclusive that the person is highest when the recall is lowest (start) and then there is a slight downward trend in precision when the recall is increased.

Each method follows this trend except for “Correlation”, its precision starts unusually low in comparison to the other method.

From this we might suggest that “Chi-Square” and “Bhattacharyya” have the best Precision-Recall as their corresponding values start high relative to the other models and finish with about the same precision as the other models (there is no definitively superior model according to this test when the recall is 30 images).

In addition, the graph suggests that as recall increases, the difference between the models decreases. This means that if the recall was increased even more the difference between the models would be reduced so that they could be indistinguishable.

**Database Read/Write Results**

There are many reasons why I should be storing data externally for this application.

This includes data backup and recovery. External storage facilitates data backup and data recovery it in case of a hardware failure, power outage, or any other unforeseen event that might cause data loss. Moreover, in terms of scalability, external storage can be easily scaled up or down depending on the needs the application. As the database grows, more storage space can be added without having to worry about limitations or slowing down the system.

Also for this application improved performance is very important. By storing data externally, this will significantly improve the performance of the application. Keeping a database of image features, so that they do not need to be calculated at every run time will substantially improve the speed of this application as the bulk of the processing time is spent calculating the 3D HSV array from each of the images in the database.

Therefore creating a file which can store this information would be very beneficial. The application will have a button on the frontend that will offer the user to update the database. This button will call a function that will calculate all of the 3D HSV arrays and store it externally. This means that for every run time the application will read the data from this file instead of calculating it every time.

At this stage it is useful to try a range of different techniques on how to do this so that the application can adapt the best performing method. The best performing method should offer the fastest read/write speeds.

I initially tried to use Microsoft SQL Server Management Studio. This meant creating a database and tables, using my local device (laptop) as a server which I could connect to in the Java application. Doing this my function for saving data would have used SQL statements to query the database and fetch all of the required data. The application would have connected to the database by using SQL Server Authentication when I create a user and a login for it.

Below is a screenshot of Microsoft SQL Server Management Studio showing the server with the database named “SearchEngine” in the Object Explorer section.

Graphical user interface, application

Description automatically generated

Through further research I found that for the current scale of this project using this method would be overkill and is not best suited to storing relatively small amounts of data.

…

I then started exploring other options. CSV then .txt. These are text-based file formats that are human-readable and can be easily edited with a text editor. This can be useful if it is needed to modify the contents of the file manually, without running the application. With both of these methods I came across the same issue which is that file took an incredibly long time to update. The having run the update function on both, here are their results below

**.txt Format**

**CSV Format**

|  |  |
| --- | --- |
| **Run Number** | **Time Taken (mm:ss)** |
| 1 | 45:12 |
| 2 | 43:56 |
| 3 | 47:43 |
| 4 | 45:43 |

|  |  |
| --- | --- |
| **Run Number** | **Time Taken (mm:ss)** |
| 1 | 44.21 |
| 2 | 45.55 |
| 3 | 44.01 |

These results show an average of 45min 38s for CSV file format, and an average of 44min 45s for .txt file format.

Text

Description automatically generatedBelow shows an example code snippet of how I initially tried saving and reading data to a CSV file name “scores.csv”.

Text

Description automatically generated

This time is way too long. So I did some research in to finding a more time-efficient method. SER and bin files

What is SER?

SER (Simple Extensible Rendering) is a binary file format that is designed for storing scientific data, including 3D arrays. It has a simple header format and supports compression, which can make it efficient for storing large datasets.

Results.

-Expecting the performance to be much better than CSV and .txt file formats I decided to store the image database, the 3D HSV array, and the image directory references. The results from this are shown below:

Update database (saving SER file): 43 minutes

Loading SER file: 46 minutes

It is the time splits here than are most interesting. Saving the image database and the image directory references took milliseconds each but storing the 3D HSV histogram array took 99+% of the time (this includes calculations and converting the 3D “arraylist” into a serializable format). Due to these less than impressive results I decided to try using the .bin file format.

What is .bin?

Bin is a generic binary file format that can be used to store any kind of binary data, including 3D arrays. It is often used when performance is a concern, as binary files can be read and written more quickly than text-based formats. Therefore, would be suited to this application. The .bin file is being used to store only 3D HSV histogram array, as shown in the code listing shown below. This method shows how a new file “src//test.bin” is created and the array is looped through each dimension as it increments through the data writing each element to the file. It is important to note that “imageHistoMatrices” is a 3D float arraylist which contains the histograms for each image. Its declaration can also be seen below.

Text

Description automatically generated

Loading the data using this method yielded much better performance results than methods trailed previously. Below shows a set of results seeing how it performed over a normal range of histogram bins, and cross validated.

|  |  |
| --- | --- |
| **Num Bins** | **Time (s)** |
| 3 | 15.0725753 |
| 4 | 15.9499971 |
| 5 | 15.5880655 |
| 6 | 13.6463177 |
| 7 | 13.9659532 |
| 8 | 14.4504188 |
| 9 | 17.2094585 |

|  |  |
| --- | --- |
| **Num Bins** | **Time (s)** |
| 3 | 13.713485 |
| 4 | 14.980522 |
| 5 | 14.302992 |
| 6 | 16.169821 |
| 7 | 13.627377 |
| 8 | 14.803532 |
| 9 | 14.556957 |

|  |  |
| --- | --- |
| **Num Bins** | **Time (s)** |
| 3 | 17.0071149 |
| 4 | 19.9838624 |
| 5 | 14.950811 |
| 6 | 18.8896121 |
| 7 | 18.0683443 |
| 8 | 19.2009816 |
| 9 | 19.3936016 |

Averages:

|  |  |
| --- | --- |
| **Num Bins** | **Time (s)** |
| 3 | 15.26439 |
| 4 | 16.97146 |
| 5 | 14.94729 |
| 6 | 16.23525 |
| 7 | 15.22056 |
| 8 | 16.15164 |
| 9 | 17.05334 |

According to the average results shown above, the trendline in the graph below shows a slight increase in the processing time when the number of bins is increased. However, this increase is very small when increasing the bins by a small amount.

These results are very good and a loading time of around 15 seconds shows a 186x improvement on the loading time from using the SER file format (46 minutes).

The code snippet shown below shows how the data is loaded into the system from the function just covered. This uses the same strategy as to when the data was written to the file because it loops through each of the dimensions of the array, iterating and storing the data in a new 3D float ArrayList.

The times of performance can be seen in the sections already covered, as this is the final function that was implemented for this application.

Text

Description automatically generated

**Error handling**

**Possible improvements**

**Conclusion**

**References**

*Correlation and Bhattacharyya have the best results in terms of accuracy to the label associated to the search image and image results (Crowds, F1 Cars, Horses, and Landscapes).*

*Averages of 65% and 70% respectively show a strong baseline of accuracy because these functions have no object detection so I would have initially guessed the results may have been lower because these methods are only dealing with colour. This means that from the test images above you can see that even the images that do not match the query image category they are still very similar in terms of colour so it would make sense for even my final search model to return images like those shown.*

*From here I should try to find a way to implement both of these methods with an object detection function.*

*The results from Chi-square and Intersection were very unimpressive in terms of label accuracy and returning images with similar colours. They will not be used in future models.*

In this example, the compareHistograms method takes two histograms as input, normalizes them, and returns the correlation coefficient as the similarity score. The correlation coefficient is computed by first computing the mean and standard deviation of each histogram, and then computing the covariance between them. The resulting score is a value between -1 and 1, where a score of 1 indicates a perfect similarity and a score of -1 indicates a perfect dissimilarity.

**Title Page**

**Into**

**GUI overview**

**Correlation**

Simple Model

Bin optimization (accuracy and time)

Cross validation

Summary

**Chi-square**

Simple Model

Bin optimization (accuracy and time)

Cross validation

Summary

**Intersection**

Simple Model

Bin optimization (accuracy and time)

Cross validation

Summary

**Bhattacharyya**

Simple Model

Bin optimization (accuracy and time)

Cross validation

Summary

**Overall results**

**Database read/write results.**

Methods tried (CSV, .txt, SER, bin)

Results (time)

**Error handling**

**Possible improvements**

**Conclusion**

**References**