COS 429 Assignment 2

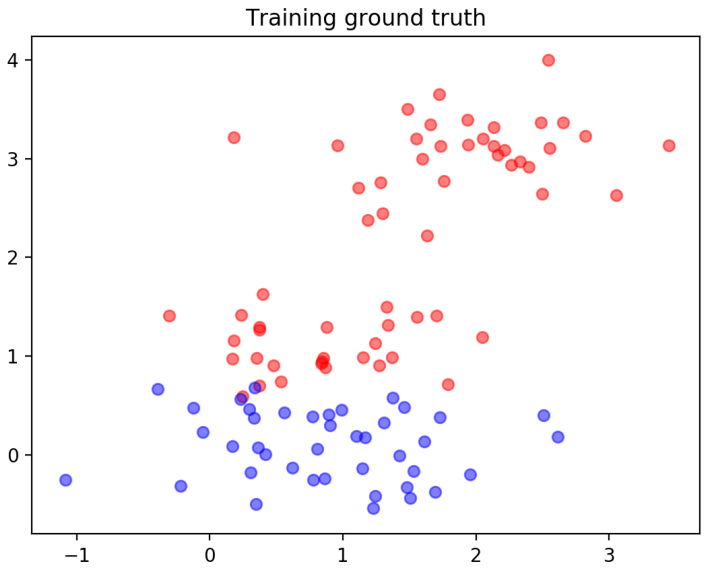
Preeti Iyer and Matthew Hetrick

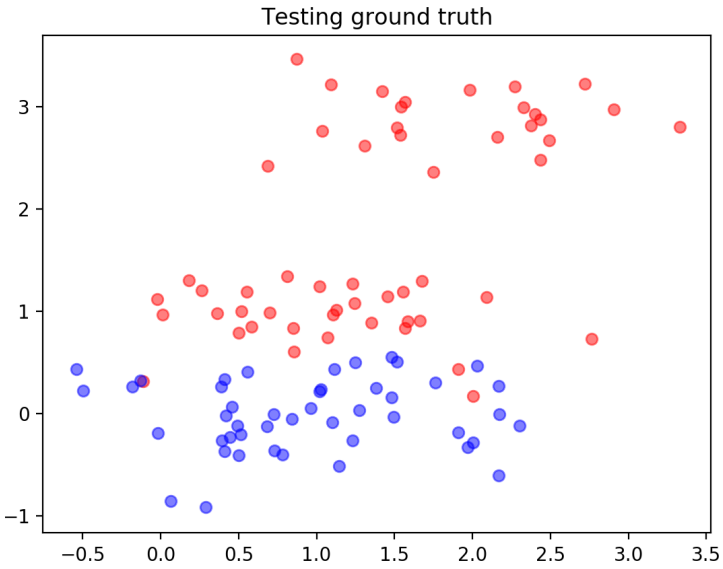
December 14, 2019

Question 1:

The classification accuracy with the linear regression model was: 0.89 The classification accuracy with our implemented logistic regression model was: 0.98 It makes sense that our logistic regression model gives a better accuracy since this model gives less influence to points far away from the decision boundary and thus limits the influence of these far away points on the overall model allowing for a more accurate fit.

The generated plot for learned predictions on training and test data with logistic regression can be seen below:





Question 2:

Part a:

See code

Part b:

The generated plots for training our classifier on 250 faces and 250 nonfaces and testing on 100 faces and 100 nonfaces using 4 orientations that are wraps at 180 degrees can be seen below:

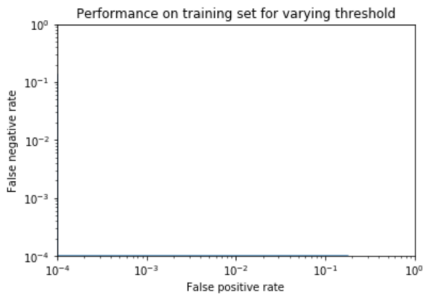


Figure 1: perfect training performance

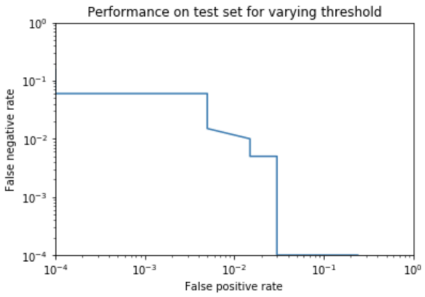


Figure 2: imperfect testing performance

Part c:

With the same test set, we ran the classifier with a much larger training size. Since this model is not over fit, in the training plot the model does not have perfect performance. But training on much much more data did improve the accuracy of our classifier; we can see this since in the testing plot is closer to the bottom left corner – it extends less in the direction of higher false positives and negatives.

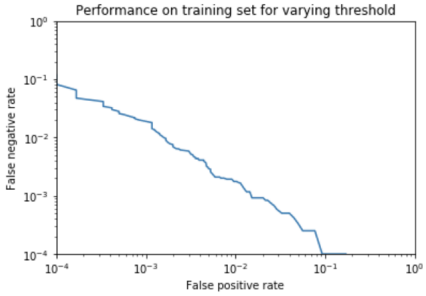


Figure 3: perfect training performance

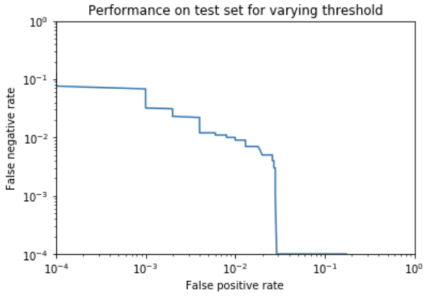


Figure 4: perfect training performance

Part d:

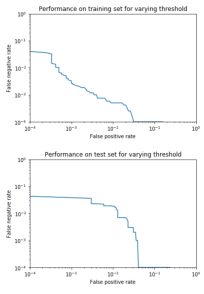
As we increased the number of orientations from 4 to 6,9,12 the testing accuracy improved very slightly (with our false negative rate in particular going down between 4-6 orientations to 9 and 12 orientations). However more notably, our curves got smoother (less step-wise).

The Histograms of Oriented Gradients technique counts occurrences of gradient orientation in localized portions of an image; the orientation of the gradient elements is accumulated into neighboring bins. Increasing the orientation thus allows for finer structure of patterns to be cauterized and understood by the descriptor, and thus increases the accuracy (roughly decreasing the false negative and positive rates) of our classifier. In the Dalal and Triggs paper it was seen that performance does not really improve beyond about 9 orientations, we expected our results to differ since face detection varies from pedestrian detection (we were thinking that the improved details from additional orientations would make a larger difference with faces because of the relatively uniform structure compared to pedestrian detection).

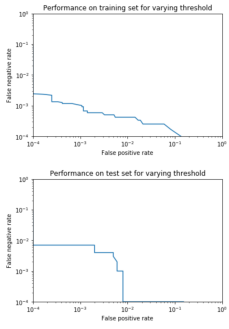
Part e:

The original domain of the arctan function times the multiplier is [-orientations, orientations], which we are mapping to [0, orientations) using the modulus operation with orientations. This is essentially mapping from a circle to a semicircle using the multiplier and the modulus and making sure that opposite radial measurements map to the same gradient direction. What we want to do is come up with a multiplier that makes is such that the arctan measurement multiplied by the multiplier gives us a [0, orientations) domain, so that the modulus operation outputs unique values for the mapping. This basically "turns off" our 180-degree wrapping. Thus, we design a wrapper such that if the wrap180 is passed in as false, we divide our orientations by 2π to obtain our multiplier. This multiplier then to turn the arctan measurements into unique values with the domain [0, orientations).

Disabling the wrapping had consistently better performance than with the wrapping, and in general the difference was about an order of 10. This is consistent from the Dalal and Triggs paper, which theorized that having signed gradient directions decreased the performance. This is most likely due to the fact that even though the paper regarded pedestrian detection, whereas for this detector we are only regarding faces, the signed gradient direction has an effect on the measured error. This is not super significant though, as the performance was only marginally better without the wrapping. See below for the output error graphs for detectors run on 12000 training images, 1000 testing images, and 9 orientations.



(a) Error with wrapping



(b) Error without wrapping

Figure 5: Error Plots for hog36 with and w/o wrapping

Part f:

We would want to run the detector with a threshold that favors a balance between false negatives and false positives, but we would want to air on the side of caution when it comes to the false

negatives; we would like to optimize the balance to have more false positives than false negatives, because we want to ensure we get as many faces as possible, even if we over-classify entities as faces rather than under-classify these faces. We thought of this because of the applications of face detection in every day use cases - let's say in the example of a snapchat filter: it's worse if the filter doesn't recognize a face than if it also misclasifies a background space as a face while also capturing the main user's face; similary for other use cases it's primarily important to capture all faces rather than focusing on optimizing a false positive rate as low as possible. Ultimately, our classifier should have some false positives and minimal false negatives.

Question 3: Single-Scale Face Detection

Part a:

See code

Part b

We tested our detector on a number of images to understand its performance - thinking carefully about false positives and false negatives. Once we saw that our detector was detecting the majority of faces in high quality and high contrast images while minimizes false detections we ran the automated testing script to look through all images.

Below we can see the results for **two well classified images** (where the detector performed well); in both of these images the contrast between the faces and background space is pretty significant and the faces are aligned head on with the camera which might be why our detector was able to perform so well on them :

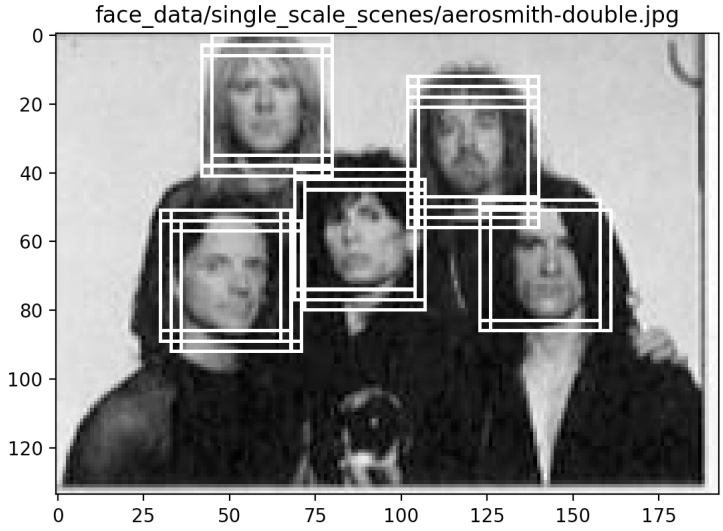


Figure 6: good detector performance : example 1

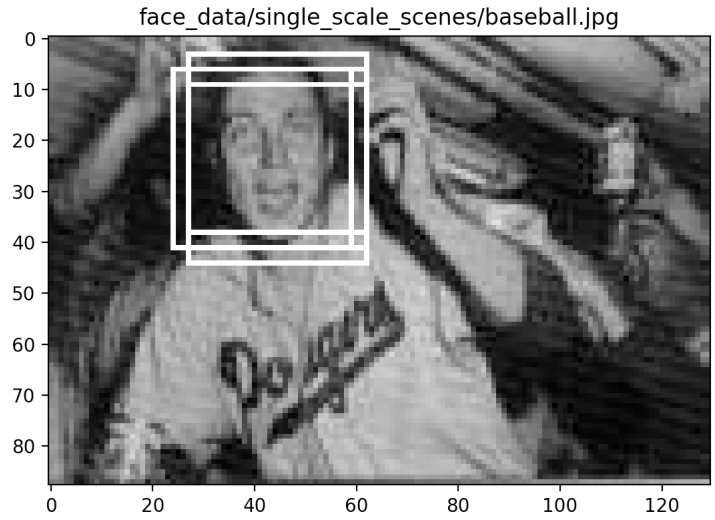


Figure 7: good detector performance : example 2

Below we can see the results for **three poorly classified images** (where the detector did not perform well); in these images there are a lot of false negatives from low contrast spaces where the distinguishing gradients between a face and the background might be hard to detect. In examples 1-2 the bending poses of the first row's knees/shins with the soccer socks resemble a pattern that might be perceived as structurally similar to some components of a face leading to many false positives :

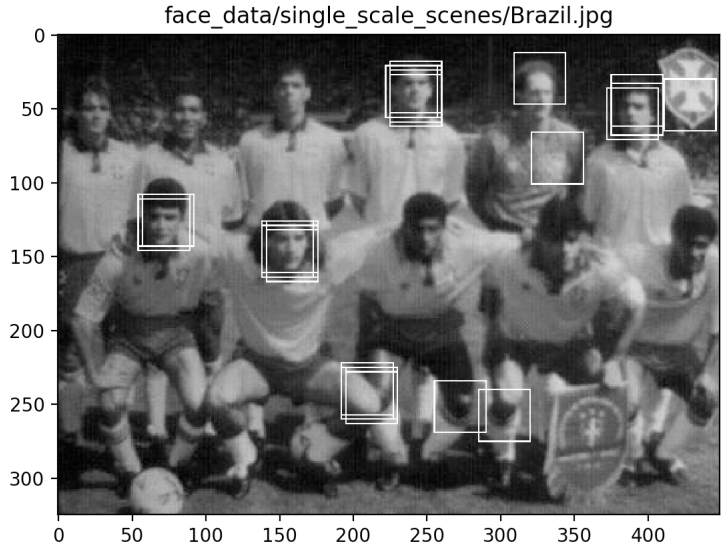


Figure 8: bad detector performance : example 1

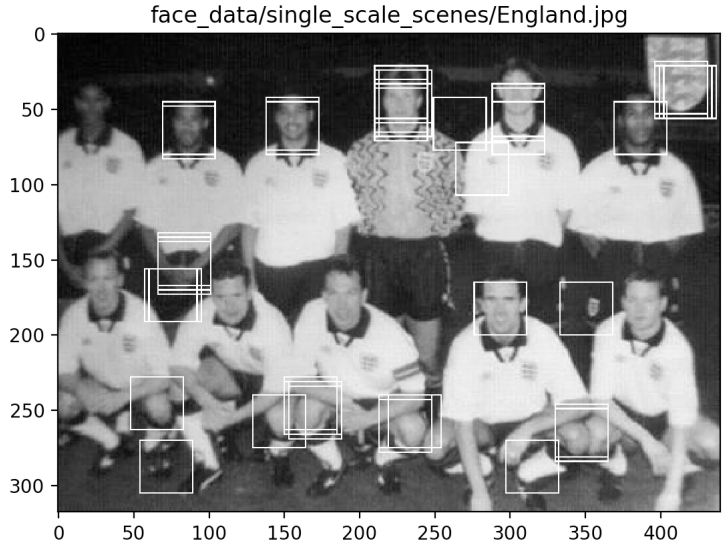


Figure 9: bad detector performance : example 2

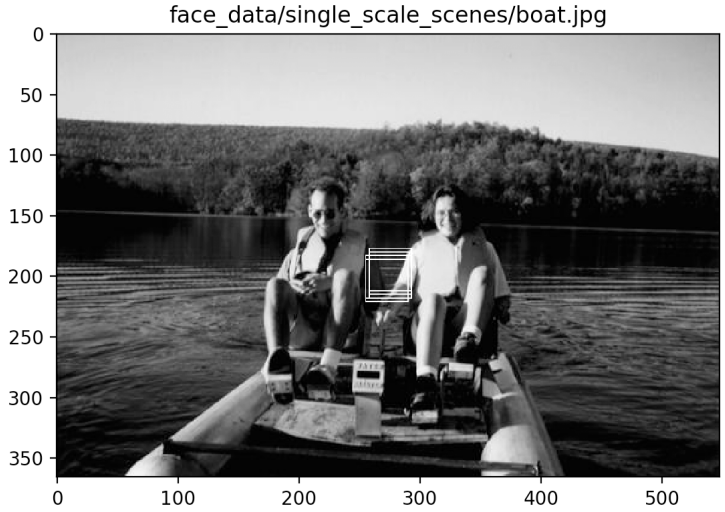


Figure 10: bad detector performance : example 3

Question 4: Face Detection and Model Fitting

Part a:

See attached code. We implemented option 2. It is important to note that the run time of this algorithm is $O(n^2)$, so input images with minimum dimensions greater than around 600 pixels took significant time to detect. But what was nice was the time sped up the farther into the algorithm we got, which is just a product of the larger window size.

Part b:

Below we can see the results for **two well-classified images** (where the detector performed well); like in part 3, in both of these images the contrast between the faces and background space is pretty significant and the faces are aligned head on with the camera which might be why our detector was able to perform so well on them. Also, the second image did a particularly interesting thing and did not classify the weird alien-looking face, which indicates that our detector is working properly specifically for human face detection:

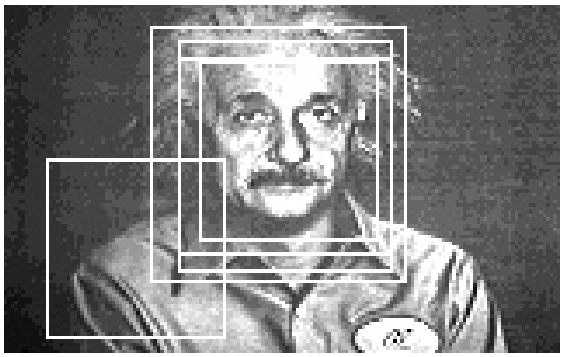


Figure 11: good detector performance : example 1

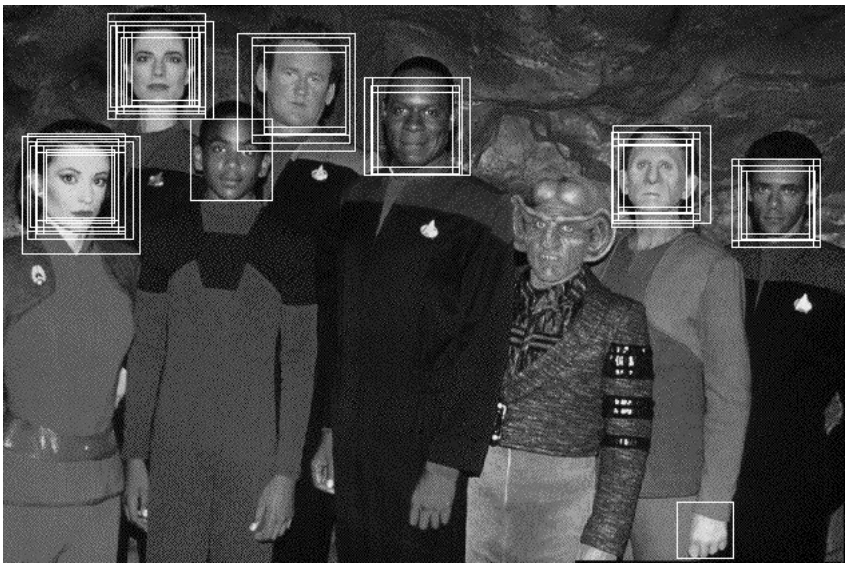


Figure 12: good detector performance : example 2

Below we can see the results for **two poorly classified images** (where the detector did not perform well); like in part 3, in these images there are a lot of false negatives from low contrast spaces where the distinguishing gradients between a face and the background might be hard to detect. Also, these images had many false positives. The window image had many windows detecting ”faces” to the upper left of the actual face, and the low-contrast image of the man’s face mainly detected his mouth as his entire face. It is important to note though that in the image with the man, the larger window did in fact detect his face. Therefore, this ”poor” classification could potentially be trained to be a ”good” classification with the appropriate window size. Here are the images:

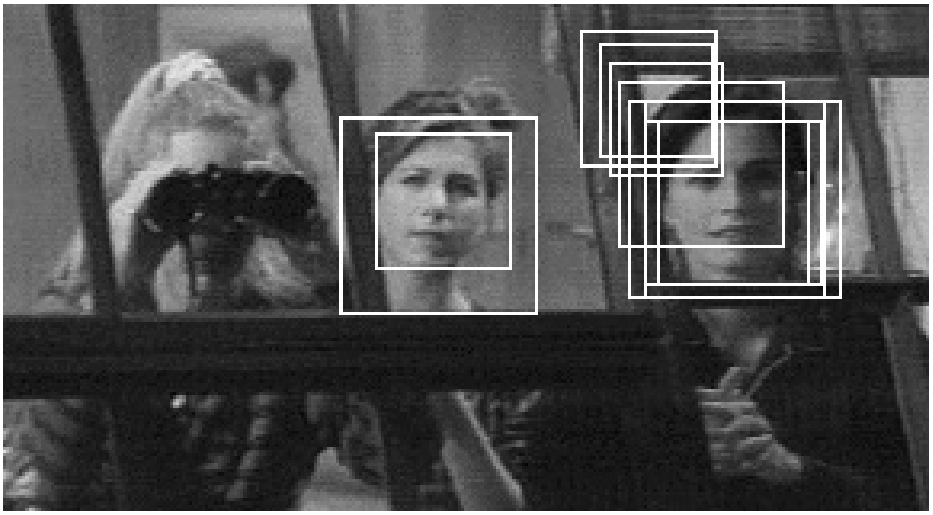


Figure 13: bad detector performance : example 1

Figure 14: bad detector performance : example 2

Extra Credit part d:

The results in $testing_scenes_boxes.txt$ contain the min and max x,y coordinates for each face, thus are boundaries that represent the pixels that would be marked white in our produced outimage. To compare our detector's results with the results stored in there we would look get the indices the pixels that are marked as a face in our outimage (those with value 255 indicating a while line to be drawn there); as we increment through our image we are keeping track of where (i,j or x,y) where are in the image so we can check if $x_{min} = i$, $x_{max} = i + windowsize$, $y_{min} = j$, $y_{max} = j + windowsize$ (in actuality we would probably code in a buffer of a few pixels) to check between our face detections and the ground-truth locations of faces in the testings scenes. If we iterate through all of the contents of the ground-truth locations and the detected faces, we can also keep track of the relative number of faces each detected and the number of our faces that meet the ground truth locations vs. the ones that don't. Understanding all of that together our evaluation function will be able to see that false negatives / positives might be higher based on the threshold we set. Understanding our threshold, false positives, accurate classifications (ground truth locations compared to our locations), and false negatives together will give us a good hollistic understand of our detector's performance in reference to the ground truth detections stored in the testing scenes bboxes file.

Question 5: Model Fitting: A Concrete Example

In this part, we were given the data set below and asked some questions regarding classification.

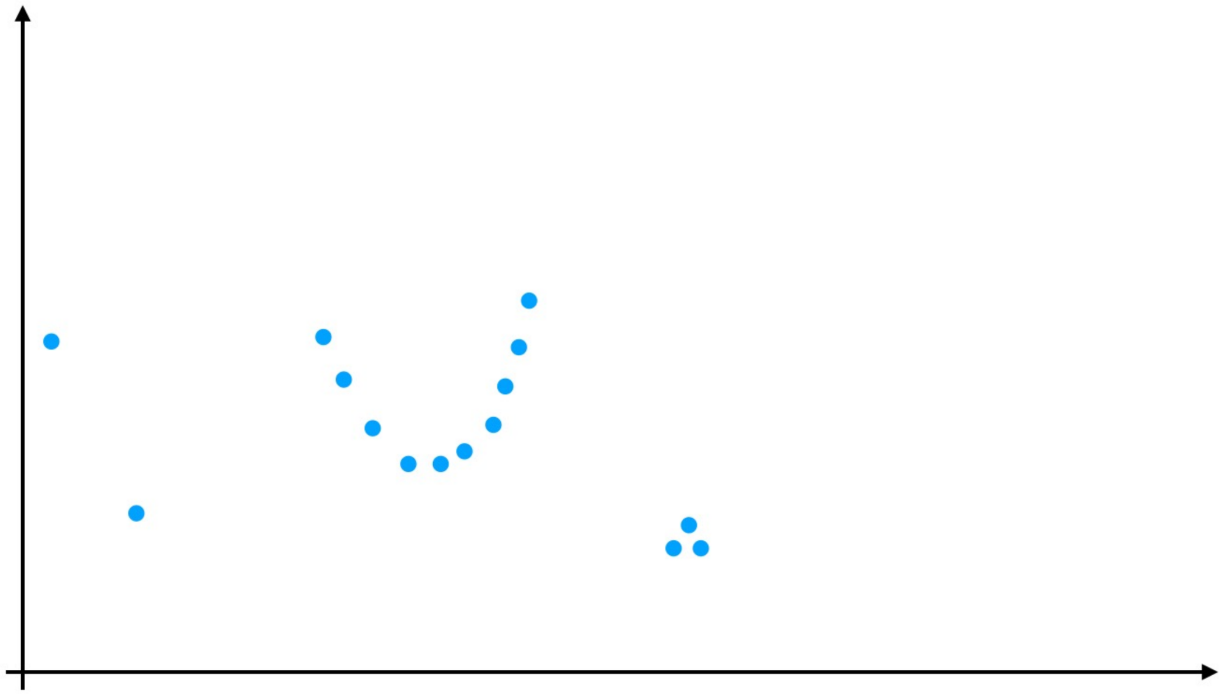


Figure 15: Data Set for Question 5

Part a:

The best class of models for this data set is a line, specifically a parabola. $y = ax^2 + bx + c$. This equation has 2 degrees of freedom.

Part b:

Sketched below is an approximate best-fit parabola. It would hover closer to the right-hand side given the higher concentration of points on that side. Note that this fit assumes a total least squares fitting model, which is not particularly robust when it comes to rejecting outliers.

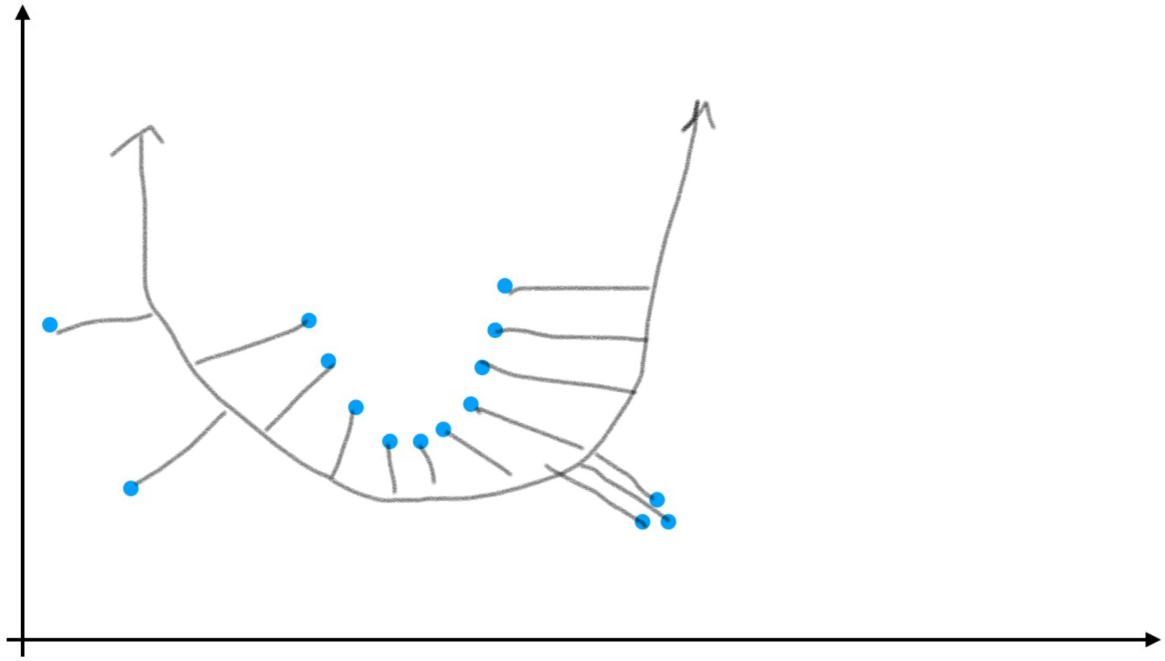


Figure 16: Approximate best fit model

Part c:

Our approximate best fit model appears visually to have about 12-13 inliers. This is from the left two points being the farthest from our best fit line, as assuming the one of the 3 points in the bottom right cluster is past our inlier threshold, we can assume 12 inliers and 3 outliers for this parabolic fit for this data set of 15 points. Moreover, we need 3 points as the minimum number of viable points to fit our parabolic model. Therefore, the initial number of points, 's', we would use is 3, and the consensus set size, 'd', we would use is 12.

Part d:

The probability of selecting an inlier is $\frac{12}{15}$, because in our model described in part (b) we assumed 12 inliers out of 15 data points. Logically, if we select data points at random without replacement, our probability for selecting 3 inliers on the first try is $\frac{12}{15} * \frac{11}{14} * \frac{10}{13}$. Note that if the number of data points (15) was much greater than the number of inliers, we could model this with replacement. But for this situation, our model has too few points to do that. Continuing, this probability can also

be written as the number of ways to choose 3 inliers from 12 total inliers divided by the number of ways to choose 3 points from all 15 data points: $\binom{12}{3} / \binom{112}{3}$ This simplifies to the logical multiplication we came up with before, and the overall probability is $P(1Trial = 3Inliers) = p' = .4835$.

Part e:

To get the number of iterations, we must derive an equation that yields a probability of convergence for a given number of iterations. We do this with the following equation: $1-(1-p')^n$ where n is the number of trials. This follows from the following logic: the probability of getting all inliers in one trial is p' as calculated above. Therefore, the probability of not getting all inliers, aka having $>= 1$ outlier is 1-p'. Then, the probability of having at least one outlier in n consecutive trials is $(1-p')^n$. Then, the probability of not having at least one outlier in at least one trial is $p = 1-(1-p')^n$.
We can then rearrange this probability equation to solve for the number of iterations. We get $n = \frac{\log(1-p)}{\log(1-p')}$. Plugging in .99 for p and our calculated .4835 for p', we get $n = 6.96$ 7 iterations to be 99% confident that at least one of the iterations has all inliers. See jupyter notebook screenshot attached to part f for python output.

Part f:

Now, instead of 15 data points of 12 inliers and 3 outliers. We increase the number of outliers to 100, so our total number of points is 112. Therefore, $p' = \binom{12}{3} / \binom{112}{3} = .000965$. We then compute n to be: $n = \frac{\log(1-.99)}{\log(1-.000965)} = 4768.7$ 4769 iterations to be 99% confident that at least one of the iterations has all inliers. Here is the code for parts e and f:

```
In [1]:  from scipy.special import comb
         import numpy as np

In [2]:  # original case
         num = comb(12, 3, exact = 'false')
         denom = comb(15, 3, exact = 'false')
         np.log(.01)/np.log(1-(num/denom))

Out[2]:  6.970012425210888

In [3]:  # 100 outlier case
         num = comb(12, 3, exact = 'false')
         denom = comb(112, 3, exact = 'false')
         np.log(.01)/np.log(1-(num/denom))

Out[3]:  4768.653356982768

In [ ]:
```

Figure 17: Python code for 5.e and 5.f

Part g:

Assuming we are still using a parabola as the model, the Hough transform space would be 3D, with the 3 dimensions corresponding to a, b, and c in $y = ax^2+bx+c$ (not assuming this, we would have

a line with m and b as the dimensions like in lecture slides). Therefore, as single point in the Hough space corresponds to a parabola in cartesian space, and a point in cartesian space corresponds to a line in Hough space. So, the Hough transform picks the point (a,b,c) in Hough space that has the most intersecting lines (which are the Hough space representations of our data points) and picks those as the parameters to fit the $y = f(x; a, b, c)$ model.

Part h:

We do not think that these two, RANSAC and Hough transform, would find the same optimal model. We think this because the parameters are found using different methods. The Hough transform works via what we described above in part g; RANSAC operates iteratively by randomly selecting 3 points with which to fit a $y = f(x; a, b, c)$ model, and then calculates the inlier ratio for that model after selecting inliers for the model based on the error function between each point in the data set and the fitted model. It then picks the model with the best inlier ratio. These two methods may pick similar optimal models, but most likely the fits will be different because we are minimizing a objective (distance) function to fit parameters and then using an inlier ratio to vote on models with RANSAC, but we are voting on models to select the parameters with Hough transform.

Moreover, the number of parameters needed to be input to the RANSAC model is much greater than for the Hough transform, as we must provide some s, d, t, and N on which RANSAC can operate. Therefore, the output of RANSAC depends on these user-defined parameters. With the Hough transform, the number of parameters is just the number of paramters with which we are trying to fit a model. Therefore, in this case the overall schema used to fit (a,b,c) is much different between RANSAC and Hough Transform and the two will most likely output different optimal models.

Part i:

If we were to run the Hough transform under the conditions in part f, we would expect it to be faster than RANSAC. This is because with so many outliers, RANSAC will take a long time out output its fit. However, this fit will most likely be more robust that from the Hough transform because of its error function optimization. The Hough transform would also probably have relatively poor output for such a noisy image, as our buckets would be quite ”shallow”, aka the Hough transformation lacking many line intersections. Additionally, one can think of this data as having multiple models, such as a combination of a parabola, a line, and a triangle. In this case especially, the Hough transform would be optimal to fit the data. Ultimately, in the case of 100 outliers, the Hough transform would run faster, but RANSAC's optimal model would be a better fit to our data set.