CSE 515 Project Phase III

Zackary Crosley, Tirth Shah, Tithi Gupta, Dhiren Tejwani, Tithi Patel, Riddhi Patel

***Abstract* — The versatility of graphs to model real world phenomenon and systems makes them incredibly important for data analysis and storage. Everything from biological systems to web navigation are often modeled using directed and undirected graphs. This paper introduces an implementation of several graph tools and algorithms. We create a graph from real world data and is formed into clusters based on relational properties. Additionally the PageRank algorithm is implemented and used for identifying dominant images, related images to an input, and for classification. Finally, a Locality Sensitive Hashing technique is used for identifying similar images.**

***Keywords*— graph, hashing, clustering, spectral clustering, PageRank, classification, locality sensitive hashing, adjacency graph.**

I. INTRODUCTION

A. Terminology

Authority: A node of a graph which is pointed to by many hubs, indicating an expertise in a topic of those hubs.

Adjacency Matrix: A matrix with the same items on both axis whose corresponding value indicates the presence or lack of an edge. Usually a value of 1 indicates an edge exists whereas a value of 0 indicates no edge.

Classification: The process by which nodes are assigned to groupings based on sample data from each group.

Clustering: The process by which nodes are assigned to groupings based on the relationship between that node and other nodes in the graph.

Cosine Similarity: It is the similarity measure, used to identify the most similar images or locations or users as per the given ID. It performs the dot product between the two vectors and normalizes these vectors by the unit length. It results into the score range [0,1] after normalizing.

Degree Matrix: A diagonal matrix D= diag(d1,…,dn) corresponding to a graph that has the vertex degree of di in the i^{th} position.

Fiedler Vector: The eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix of a graph.

Hub: A node of a graph which points to many authorities, indicating the node is a resource of aggregated information.

Locality Sensitive Hashing: An indexing method which projects data points onto a series of hashes across a series of layers to determine what groups it belongs to. Vectors for projection may be generated from the data or random.

Laplacian Matrix: It is a matrix of a graph formed by the difference of the degree matrix and the adjacency matrix of the graph.

Page Rank: A methodology for ranking the importance of a node in a graph using a simulated random web surfer moving between pages. The more times a page is encountered by the random surfer, the more central and thereby important it is.

Personalized Page Rank: A personalized version of the Page Rank where the personal interests of the user is given more importance and the links which are relevant to the user interest are given more importance.

Similarity Matrix: A matrix with the same items on both axis and whose corresponding value indicates the similarity of those two items. This (ideally) results in a matrix with one values on the diagonal, where similarity between identical items are encoded, and values between 0 and 1 filling out the rest of the graph.

B. Goal Description.

The goal of this project is to produce a program which demonstrates ability to perform analysis on graphs using a variety of algorithms. Using the dataset specified, the deliverable will create a graph and analyze it using several different methods [1]. In

specific, this entails as described for the following tasks.

For task 1 we have to create a graph from the data set specified. This entails first forming a similarity-similarity matrix between images using a subset of the data. The similarity matrix is then used to instantiate a graph using the most similar *k* images, for some user specified *k*. That graph is then displayed to the user via a pictorial or textual representation.

For task 2 we use the graph produced in task 1 to identify clusters of similar images. We implement two different clustering algorithms which identify these groups and then display them to screen using colors to differentiate cluster.

Task 3 again uses the output of task 1, this time to identify the most dominant images. That is, the images which are most central to the graph. This is done using the Page Rank algorithm, and finds the k most dominant images on the image-image graph for a user supplied K.

Task 4 using the output of task 1 applies Personalized Page Rank to find the most relevant images to three input images. These would be the images which are most dominant given those three inputs as our nodes of interest, as opposed to the entire graph like in task 3. The task outputs k images which are most relevant for a user defined k.

Task 5 uses the concept of locality sensitive hashing. In 5(a), we have to create an in-memory index structure which contains the set of vectors which were passed in input. We have to pass the number of layers, the number of hashes per layer and the set of random vectors as input. Using the distance or similarity measure, we have to find the distance between each point and project it on the input vector and store it in an appropriate hash.

In 5(b), we have to implement the image search using the index structure that was the result of 5(a) part. The combined visual model function, which contains the visual descriptors of all the models are combined with the index structure. We have to pass the image Id and 't', where 't' are the number of most similar images to the given image Id. We are expected to output the number of unique images and overall the number of images considered.

Task 6 uses the concept of two classification algorithms: a) K-nearest neighbour based classification algorithm and b) Personalized page rank based classification algorithm. We have been provided with the file which contains some of the image Ids and corresponding label assigned to it. The labels to these image Ids are to be associated with the rest of the images in the entire database and thus results into the classification of images into either of labels.

C. *Assumptions.*

- To create the image-image similarity matrix that later became the graph, we used the data from all visual descriptor models.
- The Personalized PageRank uses a maximum iteration cutoff, in addition to the threshold, to ensure completion in a reasonable time period.

II. Implementation Description

This project required several complex features and was thus separated into modules for easy organization and to encourage code reuse. This section will begin by going over the high level modules and then proceed into specific explanations of how specific functionality was implemented. Many of these modules were implemented in Phase II, and so their explanations will be shortened and focus on modifications for this phase. The high level modules that will be discussed the Interface, Loader, Database, Neighbor, and Graph modules.

The interface module provides the means for a user to interact with the code. It specifies an interface for each task to read in the appropriate inputs, validate the input, and present output (including errors). This was first implemented in phase II but improved in phase III to use Python's argparse library for a simpler and more reliable interface.

The loader module provides code to read the files from the dataset into intermediate structures before final storage. This module was first implemented in Phase II and was reused with minimal modifications. While memory hungry, the intermediate structures allowed for the separation of final data storage from the loader to prevent changes in one module from affecting the other. The loader includes several subcomponents. A generic reader class implemented recursive functionality for reading individual files from folders or even directories of folders. Since the dataset was separated into many files, this abstraction became very useful to enabling

this functionality for reading multiple different file types. Individual readers inherit from this for loading visual descriptors (descvis folder in dataset) and location data (devset_topics.xml and poiNameCorrespondences.txt file in dataset). The loader class itself uses these methods to orchestrate both the loading of the dataset and the phase III graph.

The database module stores all the data read in by the loaders into Pandas dataframes. Pandas was chosen for its speed and matrix operation. Unlike previous phases, this phase used the database only as an intermediate representation for creating the graph object, permitting a slimmer database with less functionality. The locations are stored as a single dataframe with the location information, though it is largely used for correlating names and titles with the location id. The textual descriptors are not required in this phase and thus are no longer loaded to save time and memory. Since this phase didn't require operations on individual modules or locations all visual descriptors were loaded in as written for the previous phase, but then combined once for efficiency. This simplification also enabled the dataframe to be easily saved out to a pickle file, which enabled future loads to be performed nearly instantaneously.

The neighbor module implement a knn search on dataframes using efficiencies provided by the numpy and pandas library. Using an arbitrary distance measure (provided in another file called distance.py) that would calculate the distance from a vector to reach row of a table using matrix operations, the knn would sort these outputs and filter it to the first k items. For additional speedups, the knn was implemented with multiprocessing capabilities. In multiprocessing, the table was split into a variable number of parts and each process calculated the distance from the vector to all items in the table subsection, storing them in an ordered heap. Once all processes completed the heaps would then be merged resulting in an ordered list. While this speedup seemed useful, the datasets from this project were too small for it to be necessary, and it is currently not being used. This module was implemented in Phase II and reused in this phase with small modifications for use by the LSH algorithm.

Finally, the graph module represents the similarity graph output from task 1. The graph itself uses the igraph library due to its speed and built in display functionalities. The graph module serves as a wrapper to this library, separating its inner workings from the code, making a friendlier interface, and adding functionality. This includes creating the graph object from a similarity matrix and an edge dictionary as well as cluster management functions. Additionally, the graph module includes a display function which enables the highlighting of specific nodes and coloring by cluster.

Task 1.

This tasks requires we implement a program which given a value k produces a graph of images from the Flickr data set. As stated in the assumptions section, we used the visual descriptor information across all the models as our image descriptions. These were loaded into a pandas dataframe and the cosine similarity function from scikit-learn to produce an image-image similarity matrix. To ensure self similarity is ignored, the diagonal is set to 0. For each image in the matrix, we identify the most similar k images for the user specified k and load them into a dictionary. This is then passed to the graph module which uses them to instantiate a directed graph with edge weights between nodes equal to their similarity, ignoring all connections beyond the user desired k. For efficiency purposes, our implementation saves these graphs to a pickle file for faster loading in the future. Thus our algorithm will normally loop through all k values 2 through 10 and generate these graphs for future use, returning the one of interest to the user. These pickle files can be loaded in the future using '-- graph /file/path' in the task 1 call. The graph is then displayed to the user in the form of a png file, which are created through the cairo graphics library. This image is rather large in order to display all the details of the connections. An example of one such output graph can be seen in Figure 1 below. The details are hard to determine due to the small resolution, but we can see areas of greater density and lower density in the shadings where items are more or less interconnected.

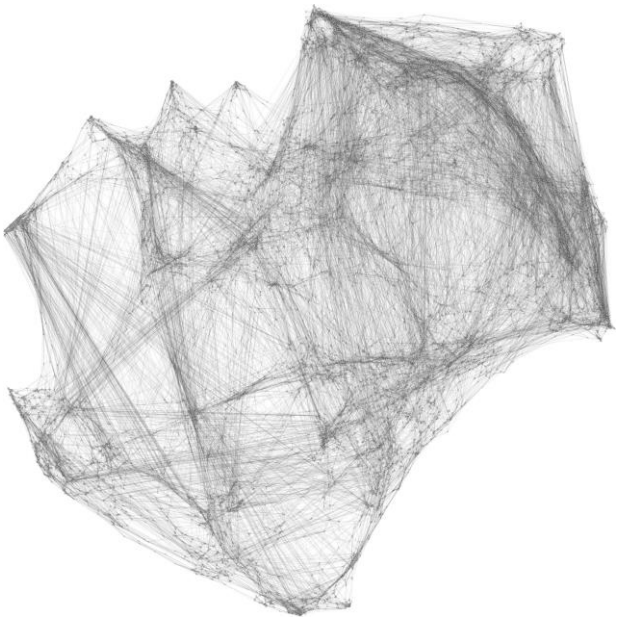


Figure 1. Similarity Graph

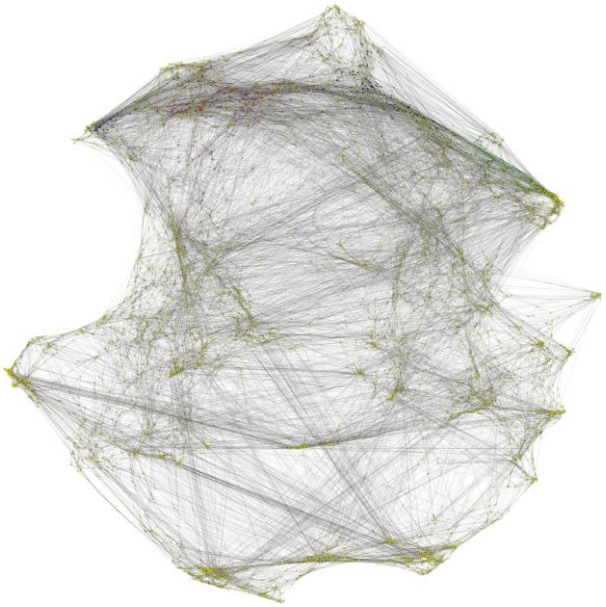


Figure 2. Six Spectral Clusters on

Task 2.

In this task, we take the same graph generated at the end of task 1 as our input. We first obtain the adjacency matrix of the graph. Given this, we take the degree matrix and compute the Laplacian matrix from the difference of degree and adjacency matrix. Now we compute the spectral decomposition.

Using the Numpy's eigen decomposition function, we get the eigenvalues and eigenvectors. Spectral clustering is all about eigenvectors belonging to second smallest eigenvalues. Hence we get the Fiedler vector of Laplacian matrix. It is then used to perform clustering. We put the images with negative values in one cluster and the rest in other.

For the second clustering algorithm, we perform eigen decomposition on the adjacency matrix of the graph. To find the clusters, we take the k most dominant eigenvectors and perform the kmeans clustering on it. This provide us k clusters as the result.

Task 3.

The PageRank score is calculated using an iterative algorithm, and it corresponds to the principal eigenvector of the normalized link(adjacency) matrix G (matrix representing state transitions Gij is a binary value representing a transition from state i to j) of the images.

A probability 's' of following a transition is taken. '1-s' is the probability of teleporting to another state. This factor is introduced to stop some images having too much influence. As a result, their total vote is damped down by multiplying it by 0.85 (a generally assumed value).

Each time we run the calculation, we are getting a closer estimate of the final value. We remember each calculated value and repeat the calculations a number of times till the numbers stop changing much. For this: 'maxerr' is taken. If the sum of pageranks between iterations is below this we will have converged. To calculate pagerank at a time account for the inlinks of state i and sink states with the s probability and teleportation to state i with 1-s probability is taken into consideration. Restart probability to all the states is assigned uniformly. The page rank of each state is normalised at the end by the sum of the pageranks of all the states. The top K (user specified) dominant images are retrieved and visualised.

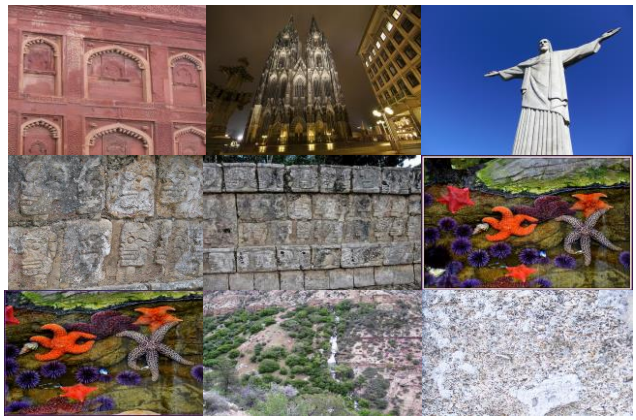
Task 4.

As in task 3, the image -image adjacency matrix is used for this task. Given 3 user specified imageids the K most relevant images are visualised using the concept of the Personalised Pagerank.

As in task 3 we take same matrix, s , 1-s probabilities and ‘maxerr’ for checking convergence of the ranks.

To visualise the relevant images to the seed images given by the user account to the in-links of the state i with probability s and teleportation to state i with probability 1-s is taken into consideration. The seed images have a restart probability of 1/length(seed vector). Here in our case the length of the seed vector is 3. Rest all the states get the restart probability of 0.

The personalised page rank of each state is normalised at the end by the sum of the personalised pageranks of all the states. The top K (user specified) dominant images are retrieved and visualised.



Task 5.

In this task, we used the visual descriptor models. We fetched the database of all the locations and their visual descriptor models. We created the data-frame containing all the image Ids of all the locations appended row-wise and visual descriptors from visual descriptor models appended column-wise. Each image is represented as the point in R^d where d is the number of dimensions or the visual descriptor values in our case. The basic problem then is to create index structure and search for images similar to query image. Number of layers, number of hashes per layer and the vectors are passed as input. We have designed layers which consists of family of locality sensitive hash functions. The idea is to hash the images using several hash functions so that for each function, the probability of collision is much higher for images which are close to each other than for those which are far apart [2]. Then we can easily determine the nearest neighbours by hashing query image and retrieving all the images stored in that bucket.

For part 5(a), we start with the input vectors that are randomly drawn using the random method with size 2xd. Now, the image point from the image dictionary is selected, then the point is mapped into R^1 by using random projections of vector. The image point with the minimum projection is selected and the distances between this image point and all the image points are calculated using the Euclidean distance. The minimum distance of all image points and the maximum distance of all image points decides the range of the hash and they are divided into the number of buckets of equal intervals. Now each image is allocated to the bucket according to its distance. Doing the same procedure for all the image points creates the index structure.

For part 5(b), we use index structure created from part 5(a) along with the combined visual model function which contains all the visual descriptors from all the visual models. We pass the image Id and the value of 't', where 't' are the most similar images to the given image Id. We use the keys of the image dictionary for comparison which contains the image Ids. We have used the knn method to find the nearest neighbours of given image Id. The output contains the image itself with distance 0 and the nearest 't-1' images with minimum distances. The number of images that are compared are also added to the result as demanded.

Figure 5. Six most similar

Task 6

In the first part, we perform K-nearest neighbour-based classification. The file containing the sample image Ids and corresponding labels is passed in as input to classify all the other images of database to either of the categories. We fetched the data-frame from the visual database, which contains the image Ids and the values of all the visual descriptors appended column-wise. We stored the sample images and their labels in key-value pair with image Id being the key. Now, for all the images which are not in the sample images file, we searched for the neighbours of all those images. We passed the stored key-value pairs, labels, index and the value of 'k', where k is the number of nearest neighbours. Euclidean distance is used as the distance measure to find all the nearest images. The images that are nearest ones increased the counter of that category by 1 and the result of count is returned. The image Id and result is appended into the existing dictionary as key-value pair. Thus, it returns all the images along with the label assigned to it as a result. We can visualize the results from the classification graph.

In the second part, we perform the Personalized Page Rank-based classification. Given the file of image Id and labels associated to it, the K most relevant images are visualised using the concept of the Personalised Page rank. The image -image adjacency matrix is used for this part. Each image Id is appended to the index. We stored the sample images and their labels in key-value pair with image Id being the key. we initialised s, 1-s probabilities. We have stored all the image Id along with their weights which has impact on the new image Id that is being classified. We constructed the seed matrix which contains the weights normalized by the number of outgoing edges of that image Id. Each image Id also has a vector associated with it which shows the probability of that image Id belonging to category. More the probability, higher the rank for that image to be classified under that category. The personalised page rank of each state is normalised at the end by the sum of the personalised page ranks of all the states. The incoming edge of image Id which has the highest weight, has the impact on the classification of that image Id.The vectors of that image Id add up to the vectors of the image Id we are classifying. The highest value in the vector categorizes the image to that category. We perform this process iteratively; each image undergoes process in each iteration. The image Id and result is appended into the existing dictionary as key-value pair. Thus, it returns all the images along with the label assigned to it as a result. We can visualize the results from the classification graph.

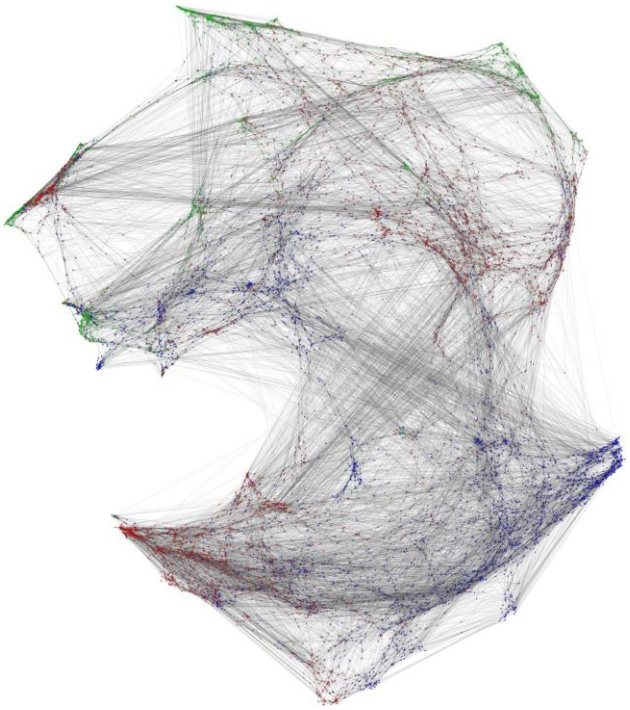


Figure 6. Three clusters using KNN

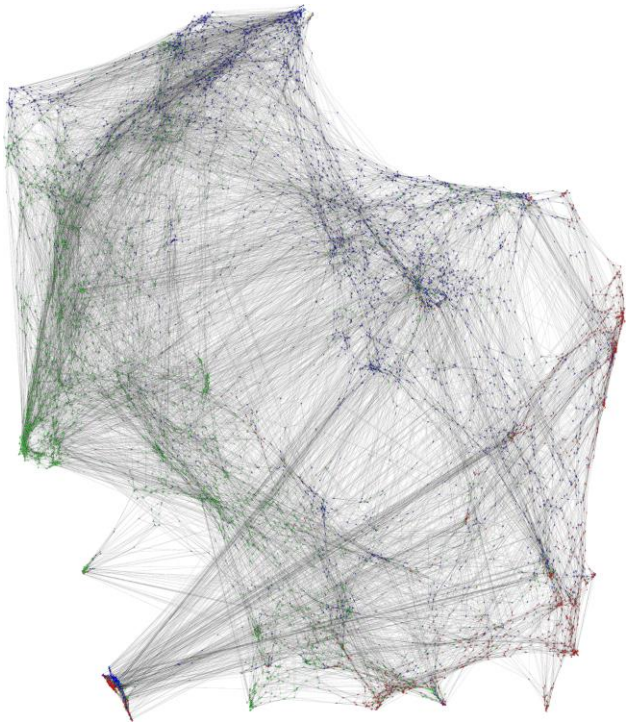


Figure 7. Three clusters using PPR

III. Interface Description

Task 1.

Command at Prompt: -task 1 --load <dataset location> --k #

Load: Location of the dataset.

K - Int number of similar images to maintain edges to in graph.

Task 2.

Command at Prompt: -task 2 --k #

K - Int number of clusters to form.

Task 3.

Command at Prompt: -task 3 --k #

K - Int number of dominant images to find.

Task 4.

Command at Prompt: -task 4 --k # --imgs image1 image2 image3

K - Int number of relevant images to find.

Imgs - list of space delimited images by int id to use.

Task 5.

Command at Prompt: -task 5 --layers # --hashes # --k # --imageId imageid

Layers - Int number of layers to create.

Hashes - Int number of hashes to create per layer for hash.

K - Int number of similar images to find.

ImageId - Int id of image to find nearest images to.

Task 6.

Command at Prompt: -task 6 --alg algorithm # --file filename

Alg - Algorithm to run.

File - The file containing the sample image Id and the labels assigned to it.

IV. System and Installation Instructions

The program is designed to run in Python 3.6.0 and above. Once Python 3.6.0 or above is installed, use the Pip installer to add the following libraries:

- lxml 4.2.5
- numpy 1.15.2
- pandas 0.23.4
- pillow 5.3.0
- python-igraph 0.7.1
- pycairo 1.18.0
    - Requires Cairo install. Instructions at https://www.cairographics.org/download
- scikit-Learn 0.20.0
- wheel 0.32.3

To execute the code, execute the *main.py* file from command line with python 3.6 (or above) and the libraries installed. (IE terminal:$ python3.6 main.py)

V. CONCLUSIONS

Our team has been able to implement the following functionality. Task one produces a similarity graph between images with a variable number of out edges for each image. Task 2 produces K clusters using a k-spectral clustering algorithm and two clusters using a standard spectral clustering algorithm. Task 3 successfully identifies the *k* most dominant images in a graph using the Page Rank algorithm for a user defined k. The images which this task produces appear to be those with specific visual properties common in other images - be it bright colors, soft grey colors, rock textures and architectural features. Task 4 integrates the personalized Page Rank algorithm to find the most relevant images given a specific set of input images. We find that the images this selects is often the most dominant images identified in task 3, especially when the images have little in common. This is expected, as the most relevant items between disparate parts of the graph will inevitably be the most dominant images. Task 5 successfully develops a hash indexing using user defined number of layers and hashes to sort images into bins. It then successfully finds the nearest *k* items to an image id, both supplied by the user. We found the success of this search is greatly dependent on the vectors used, as the possibility of unrelated items ending up in the same bin are quite high otherwise. Finally, task 6 uses classification samples to produces classes using KNN classifier and Personalized Page Rank classifier.

VI. BIBLIOGRAPHY

[1] *B. Ionescu, A. Popescu, M. Lupu, A.L. Ginsca, H. Muller, Retrieving Diverse Social Images at MediaEval 2014: Challenge, Dataset and Evaluation, MediaEval Benchmarking Initiative for Multimedia Evaluation, vol. 1263, CEUR-WS.org, ISSN: 1613-0073, October 16-17, Barcelona, Spain, 2014.*

[2] "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions" (by Alexandr Andoni and Piotr Indyk). Communications of the ACM, vol. 51, no. 1, 2008, pp. 117-122.

VII. APPENDIX

Zackary Crosley: Database modifications and Task 1. Graph and image displays for other phases. Report compilation.

Tirth Shah: Task 5 and 6 implementation and write up.

Tithi Gupta: Task 5 and 6 implementation and write up.

Dhiren Tejwani: Task 2, 3, and 4 implementation and write up.

Tithi Patel: Task 2, 3, and 4 implementation and write up.

Riddhi Patel: Task 2, 3, and 4 implementation and write up.