Phase 3 Report

CSE 515 Fall 2021

Prof. K. Selçuk Candan

Arizona State University

Group Members: Ayush Anand, Pritam De, Sairaj Menon, Sritej Reddy, Aaron Steele, Shubham Verma

Abstract: To build on the topics introduced in Phases 1 and 2 of the project and in class over the semester, image classification, indexed storage, and query refinement techniques were applied on a dataset of transformed images obtained from the Olivetti dataset [9]. Images were used to train models based on SVM, Decision Trees, and PPR to classify unlabeled datasets. Additionally, LSH and VA File index files were implemented to store and quickly query feature vectors computed on input images. Query results were then refined through user input by the labeling of images as ‘relevant’ or ‘irrelevant’. All tasks were then combined into one through a coherent user interface.

Keywords: SVM, Decision Tree, PPR, LSH, VA File, Image Classification, Olivetti, Latent Features, SVD, PCA, Index Files, Relevance Feedback

Table of Contents

Introduction

Terminology

The following terms are used in this project and paper:

- **PCA** – Principal Component Analysis, which is a method of feature decomposition which tries to preserve the variance in the original dataset. This is done by decomposing a similarity matrix into 3 separate matrices.
- **SVD** – Singular Value Decomposition, which is a method of feature decomposition which attempts to minimize the error introduced. This is done by decomposing the original data matrix and selecting some number of latent features.
- **K-means** – Clustering technique to reduce the dimensions. K-means divides the dataset into k clusters and calculates the distance between each object and the k cluster centroids which result in k latent semantics.
- **LDA** – Latent Dirichlet Allocation is a method of feature decomposition which is a form of statistical topic modeling. Here, objects are represented as a mixture of topics with each topic consisting of several features. These topics are present within a hidden layer also called the latent layer. In this project, we use the `LatentDirichletAllocation` module present in the `sklearn` library.
- **Color moment** – A method of feature extraction on images which relates to colors in an image. For the purposes of this project, it refers to creating a feature vector by computing the mean, standard deviation, and skewness of 8x8 regions in the image.
- **ELBP** – Extended Local Binary Pattern is a visual descriptor which is powerful for texture classification. It works by iterating through all pixels in the image and seeing the pattern of bytes that are darker and lighter around each pixel.
- **HOG** – Histogram of Oriented Gradients is a method of extracting features from target images. This finds gradients of change on a target image and effectively finds image contours in the ‘shape’ of the target image.
- **PageRank** – An algorithm run on graphs to determine which nodes in a graph are ‘most important’ by seeing which nodes are visited most in a random walk (with random jumps). An extension of this algorithm is Personalized PageRank (PPR), which uses a number of seed nodes to determine the importance of nodes in the graph when using the seed nodes as starting points for the algorithm. The Personalized PageRank algorithm is detailed by Candan et al., along with some improvements, in [1].
- **LSH** – An index structure described in [3] where input vectors are hashed into multiple buckets with random hashes designed to increase collisions of similar images. This is done in multiple layers.
- **VA-Files** – Vector-Approximation Files are a method of storing and indexing data for efficient retrieval and nearest neighbor queries. This is done by using a limited number of bits to represent each dimension, and splitting the original space into “blocks” in which data points are stored. This is described by Blott and Weber in [4] and [5].
- **SVM** – Support-Vector Machines (SVM) is a method of data classification which seeks to find dividing lines/hyperplanes which properly divide input into a number of labeled

classes, where points in a region are of a certain label. They are described by Drucker, Shahrary, and Gibbon in [8].
- **Decision Tree** – It is a simple and efficient method to partition a given set of data into homogeneously labeled classes.
- **Classification** – The process of taking a set of labeled inputs, grouped into several classes, and using this data to provide labels to unlabeled input images. For the context of this paper, SVM, Decision Tree, and PPR are classifiers used to assign class labels to images.

Goal Description

There were four main categories of goals for this project in the 8 tasks provided to the group:
- **Tasks 1-3** – Perform image classification on a folder of unlabeled images, assigning each image a type, subject, or image ID depending on the task. This is done by using SVM, Decision-Tree, or PPR to classify the unlabeled images based on an input folder of labeled images. False positive and miss rates for each class are computed to determine the accuracy of each classifier with different data sets and input parameters.
- **Tasks 4-5** – Create and store image feature vectors (original or in a latent feature space) in an index format using LSH or VA-File methods. For the task itself, an index should be created on an input set of vectors (with certain parameters for the index itself). Nearest neighbor search is then performed in this index structure to find the *t* most similar images to some query image.
- **Tasks 6-7** – Use Decision Trees or SVM classifiers to perform relevance feedback on queries performed in Tasks 4-5. This is done by taking user input on which images are relevant/irrelevant, then using the classifiers to refine results. The new query results are then returned to the user.
- **Task 8** – Combine the work done in Tasks 4-7 to create a full query/feedback interface. This allows a user to provide a query to an index structure (LSH or VA-File), receive results, then refine these results. This is essentially combining the work in Tasks 4-5 and 6-7 with a user interface.

Assumptions

Our group made a number of assumptions about how to process the data in the project and how various input/output files would be used in the project:
- All images (including unlabeled images or query images) are passed in in the format specified in the project parameters. This assumption is central to how data is processed and stored.
- We assumed that we did not need to implement all methods of latent feature extraction from Part 2 of the project. Specifically, only PCA and SVD are implemented as options to extract latent features.
- We assume that users will only use ‘valid’ input parameters (so very large values or negative values are not chosen).
- We assume that only one method of query refinement is necessary in Tasks 6-7.

- We assume that only latent semantic vectors will be used for classification in Tasks 1-3. This is explicitly given in the project specification, and the code in these tasks is written around this property. Tasks 4-8 are able to use latent feature vectors as well as original feature vectors.

Solution and Implementation Description

Overview

This project was implemented in Python, and all tasks are CLI-based programs which receive input from the user. Each task is in its own file and has its own documentation. Additionally, there are a number of library files which are used for functions, such as feature extraction, I/O, and miscellaneous other common functions used throughout the project.

To extract feature vectors from the images in the database, the code from Aaron Steele's implementations in Phase 1 of the project were used. They are laid out as follows:

- **Color Moment** – A vector of length 192, where the first 64 elements represent the mean color value for the 64 8x8 pixel regions in the image, and the remaining sets of 64 elements are the standard deviation and skewness of the regions respectively. These functions were implemented from scratch in Python.
- **ELBP** – A vector of length 640 which is essentially 64 histograms with 10 elements concatenated together. Each set of 10 elements is a histogram describing the number of pixels in each 8x8 region in the image which have one of the 9 possible uniform texture patterns (or none). The implementation of ELBP from `skimage` is used here.
- **HOG** – A vector of length 1764 which is a large number of histograms concatenated together. In this case, for each of the 49 2x2 groups of 8x8 pixels in the image, a histogram is created and concatenated to the feature vector. The implementation of HOG from `skimage` is used here, where the number of orientations is 9, the size of each cell is 8x8 pixels, the shape of each block is 2x2 cells, and the 'L2-Hys' block normalization technique is used.

To extract latent feature vectors from images, implementations of PCA and SVD are used from Phase 2 of the project. LDA and K-Means are not used in this phase of the project. This is due to the fact that better results were obtained in Phase 2 with PCA and SVD, and both methods were more efficient than the other methods of feature extraction.

- **PCA** – Numpy is used to find covariance matrices and eigenvalues/eigenvectors. From there, the PCA is performed in full.
- **SVD** – An iterative power method is used to compute the k latent features required, one at a time.

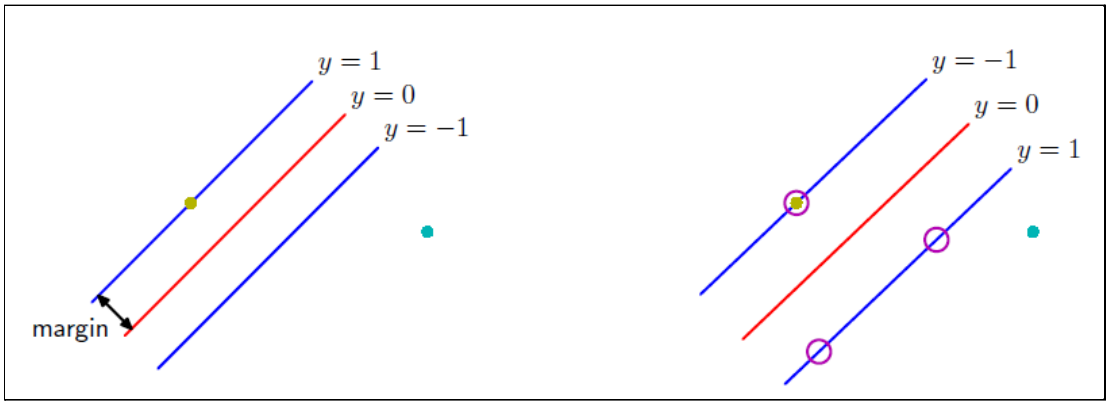
Tasks 1-3

SVM Classifier

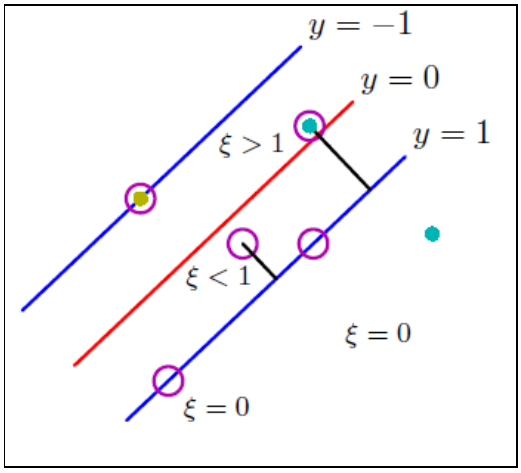
A Support Vector Machine, also known as a Max Margin Classifier, tries to find linear boundaries between objects belonging to different classes. The decision boundary is chosen to be the one for which the margin is maximized. The margin is measured by the perpendicular distance between the decision boundary and the closest of the data points. The location of the boundary is determined by a subset of data points known as support vectors.

The equation of the data points:
$y(x) = wT\varphi(x) + b$ [$\varphi(x)$ is the feature space transformation and b is the bias]

The training data has N training samples or vectors x1, x2, …., xN with target values t1, t2, .., tN Where $tn \in \{-1, 1\}$, and class labels are assigned to new objects based on the sign of y(x)



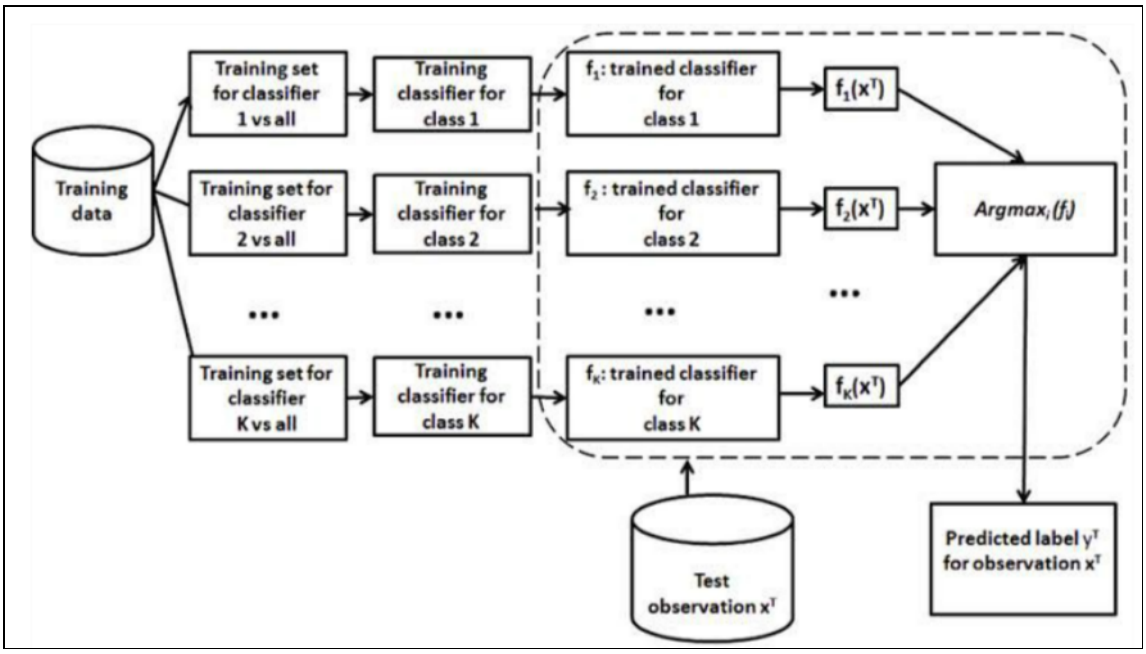
Even though we assume that training data points are linearly separable in the feature space and SVM will give exact separation, the decision boundary can be non-linear and class distributions may overlap, resulting in poor generalization. So we need to modify SVM in order to allow some misclassification. We can allow some points to be on the wrong side of the boundary with a penalty which increases with the distance from the boundary. In case of separable cases the error function will give an infinite value if a data point is misclassified or zero if it is correctly classified. We introduce a slack variable for each data point. Th slack variable is zero for data points which are on or inside the correct margin boundary and otherwise $\xi n = |tn - y(xn)|$
The slack variable is zero for the data point which lies on or inside the correct margin boundary. The slack variable is one for the data point which lies on the decision boundary. The points with a slack variable greater than one will be misclassified.



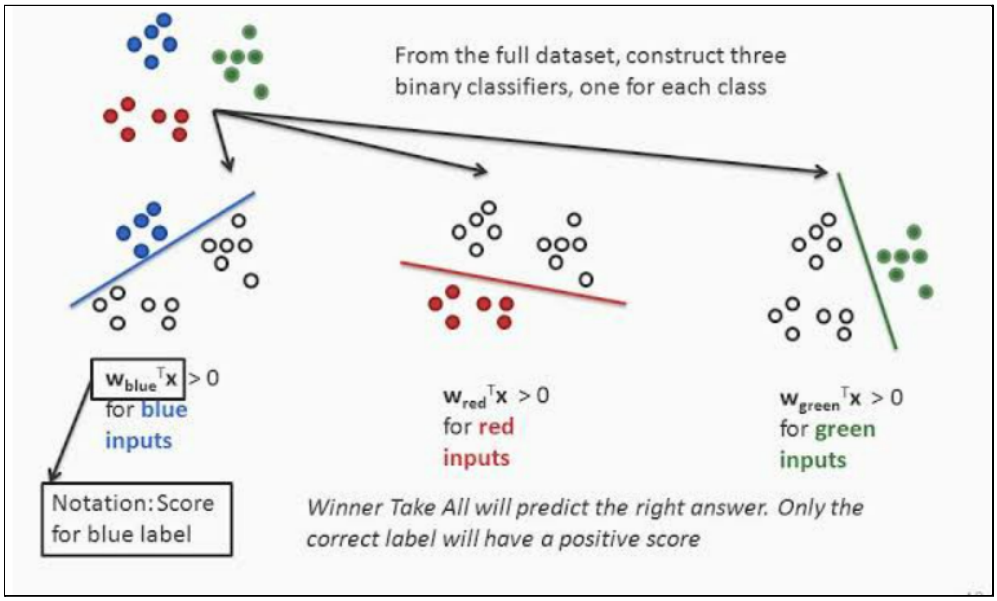
As we have multiple labels to be handled in our project (12 classes for image type, 40 classes for image subject, 10 classes for image sample ID), we need to implement multiclass SVM in order to classify the images.

We have implemented a one-vs-all classification technique. For a dataset with N class instances, we will have N-binary SVM classifiers.

Below is the data preparation workflow for one-vs-all classification:



One-vs-All Classification:



Results in the project when using SVM on Task 1 were typically decent, with results ranging from 40%-60%. However, Tasks 2-3 produced significantly worse results. This is discussed later, but the group believes this is due to the fact that image subjects and IDs are less distinguishable/separable than image types.
Please find below the accuracy results of Task1, Task2 and Task3:

Task	Feature Model	Dimensionality Reduction Method	K Value	No of Images	Accuracy
1	ELBP	SVD	5	4000	46%
	ELBP	SVD	5	3000	31%
	ELBP	SVD	5	2000	33%
	ELBP	SVD	5	1000	31%
	ELBP	SVD	5	500	54%
	ELBP	SVD	5	100	43%
2	ELBP	SVD	5	4000	5.80%
	ELBP	SVD	40	3000	4%
	ELBP	SVD	5	2000	3.90%
	ELBP	SVD	5	1000	5.80%
	ELBP	SVD	40	500	3.90%
	ELBP	SVD	20	100	4.90%
3	ELBP	SVD	40	4000	8.80%
	ELBP	SVD	40	3000	8.80%
	ELBP	SVD	40	2000	7.80%
	ELBP	SVD	40	1000	11%
	ELBP	SVD	40	500	11%
	ELBP	SVD	20	100	14.00%

Decision Tree Classifier

A Decision Tree is a simple representation for classifying examples. It is a supervised machine learning algorithm where the data is continuously split according to a certain parameter. The

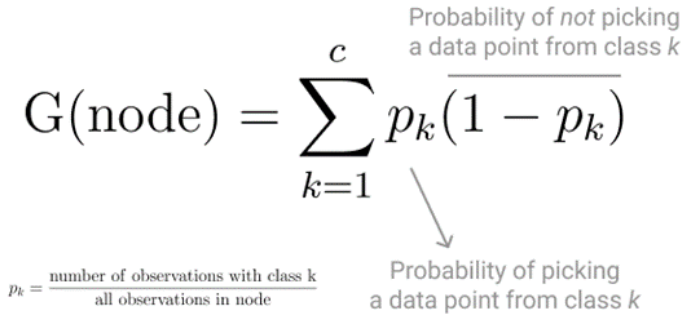
intuition behind Decision Trees is that you use the features in the dataset to create yes/no questions to continually split the dataset until you isolate all data points belonging to a class or reach another stopping condition such as maximum tree depth etc. If we decide to stop the process after a split, the last nodes created are called leaf nodes.

The class assigned to the leaf is the class of elements in that node if all the elements have been classified homogeneously. When there is more than one class present, a majority voting is done and the one with maximum representation is chosen.

After the training part is done, we test the model by bringing in data which doesn't have the label information and is assigned the label value based on the tree we built earlier.

In our decision tree implementation, we first get the training data along with the correct labels i.e. in case of task 1 we get all the images from the folder chosen for training (eg: images like image-cc-5-4.png), we extract the features from the images based on the model chosen, reduce the dimensionality using one of the chosen techniques i.e. SVD or PCA. Finally we use this resultant feature vector list along with the label extracted out of the name (i.e. 'cc' in this case) to build the decision tree.

To build this tree, we recursively iterate through the training data by splitting it based on a certain threshold value condition on one of the selected features. To do this selection, in this implementation, we calculate the maximum information gain for each of the possible threshold values. We chose the Gini index to calculate the information gain in this case. Gini Index can be defined as :



Once we find a feature,value pair where we maximize the information gain, we split the tree based on it and continue with this step until we can get to homogeneous leaf nodes that contain a single class or hit one of the stopping conditions. The stopping conditions added here are the maximum tree depth and the minimum number of classes for the leaf node. For instance, if this value is set to 3, the algorithm will stop the tree building process and assign that node a class value based on the class value of the majority of elements present in it.

In the prediction phase, we consider each input image's feature vectors and then traverse the tree based on the decision node values. Once they reach a leaf node, the label/class value corresponding to that node would be assigned to the input image.

Regarding the performance for the decision tree for the test dataset provided, tasks 1-3 were tested using all three feature models i.e. Color moments, HOG and ELBP, k value = 5, 10, 50 and different training data set sizes (100,500,1000,4000). The observations were as follows

- In the case of task 1, where we classify based on the type of the image, the accuracy ranges from around **3% to 55%**. The factors affecting the accuracy significantly was the training data size. If the dataset chosen is around the size of 1000, the accuracy measures go up. Whereas if we chose a training data set of size 500 or less, the accuracy significantly dropped to single digit with the same being noticed for data size of 4000, which would be attributed to overfitting of data. The best result was observed for the following input: k = 50, feature = hog, data size = 1000 (Accuracy – **55.87%**)
- In case of task 2, the performance was significantly worse with accuracy ranging from **2% to 6%**. Compared to task 1, the performance seems much worse because the number of labels(40 subjects) is way more than that for task 1(12). This combined with the fact that the features of type are more distinguishable than subjects, especially in cases where it has been transformed. The best result was observed for the following input: k=5, feature = color moments, data size = 1000 (Accuracy – **6.86%**)
- Finally, for task 3, the accuracy was on average a little better than task 2 with it ranging from **5% to 15%**. This performance can be attributed to the fact that distinguishing features between different facial orientations is much harder when compared to type. Hence, the classifier would end up classifying the image incorrectly to another similar orientation. The best result was observed for the following input: k=50, feature = ELBP, data size = 1000 (Accuracy – **15.68%**)

We tried to get the performance better by trying out a few ensemble approaches such as splitting the train data and training separately based on these subsets; no significant performance improvement or impact was observed. That being said, the performance could most certainly be improved by fine tuning the algorithm in the future.

PPR Classifier

No specific method of image classification was specified for PPR, so it was necessary to perform research and determine a method of using PPR for accurate image classification. Ultimately, an approach by Lin and Cohen in [2] was used to classify images, using an algorithm they call `MultiRankWalk`. Their algorithm works as follows:

- Create a graph G which includes labeled and unlabeled images as nodes, with some edges between the nodes.
- For each class c, set the seed vector s such that all labeled images of class c are assigned an equal probability $1/n$, where n is the number of labeled images of class c. All other values in the seed vector are given a value of 0. The PPR ranking R_c is then computed for the class.
- For each unlabeled image x, assign it a label c', where $R_{c'}$ is the ranking in which the unlabeled image x has the highest score (out of all PPR rankings).

The approach in [2] does not cover some crucial details of this method, however, so it was necessary to perform other tasks (such as how to create the graph initially). To create the graph used for the PPR ranking, a straightforward method was used. Distance values are computed from each node to all other nodes in the graph using Euclidean distance (which provided best results for multiple types of vectors), and the shortest n distances are preserved. If the distance from vector u to v is one of the shortest n distances, an edge is drawn between nodes u and v in the graph. However, all edges in the graph are assigned equal weight for simplicity. The graph is then stored as an adjacency matrix. Afterwards, it is converted into a transition matrix (where each edge is assigned equal probability).

To compute the PPR rankings themselves, an explicit formula was used to compute the ranking vectors. According to the work by Candan et al in [1], one is able to compute a PPR ranking with the following formula:

$$\boldsymbol{\pi} = (1-\beta)\mathbf{T}_G \times \boldsymbol{\pi} + \beta\mathbf{s},$$

In this formula, πrepresents the ranking vector, βrepresents the damping coefficient, T_G is the transition matrix for the graph, and s is the seed vector. With the following inputs, `numpy` was used to solve the ranking for each class:

```
np.linalg.solve(I - (1 - d)*T, d*s_n)
```

Although this was never encountered in testing, the result is approximated using `np.linalg.lstsq` in the rare case that the first matrix does not have an inverse.

The implementation for Tasks 1-3 is similar, where an adjacency matrix is first computed on the input vectors, then the class rankings are created (on image type, subject, or id, depending on the task). From there, the labels are computed and returned to the user. This method is not especially efficient—in particular, creating the adjacency matrix is very slow (and would be unfeasible for data sets with more than ~6000 images). However, all test runs using PPR classification concluded within 1-1.5 minutes of computation.

Some improvements could be made to this method, such as saving partial adjacency matrices or taking edge weights into account (such as assigning higher weights to edges if their distance is short). However, the simple approach taken produced good classification results, and it was not necessary to attempt these more complex enhancements to the classification algorithm. Additionally, in this project the number of neighbors from each node varied depending on the task (dividing the total number of images by 25, 50, and 20 for Tasks 1-3 respectively). This was done to account for the fact that there were differing numbers of images in each class, and provided the best results in testing, but could likely be improved upon.

A final note is that PPR classification proved to be unreliable when classes were more difficult to distinguish. Although results were often good on types of images, with accuracy results typically ranging from 65%-80%, results were worse when classifying by subjects (ranging from 45%-70%) and significantly worse when classifying by ID (accuracy from 10%-25%). This is

likely due to the fact that classifying images by ID leads to significantly less distinguishing features, making it far more difficult to accurately classify images.

For a more detailed analysis of results, some tests were run on the '`1000`' dataset with a value of k of 50 and a random sample of unlabeled images. The following accuracy results were obtained with PPR. Accuracies decreased when using smaller values of k.

- Task 1: **65%** when using Color Mode, **94%** when using ELBP, and **87%** when using HOG.
- Task 2: **67%** when using Color Mode, **33%** when using ELBP, and **61%** when using HOG.
- Task 3: **13%** when using Color Mode, **13%** when using ELBP, and **27%** when using HOG.

As can be seen from the results, PPR was the most reliable of all classifiers and typically produced high accuracy classifications.

Implementation

The implementation of each of Tasks 1-3 was very similar. First, either a latent semantic file is read into the program as input, or latent semantics are generated on a specified input folder (given via command line). If latent semantics are generated, they are saved for further use. Afterwards, the images from the folder of images to be classified are read in, and data is saved about what the 'correct' classifications are for each image.

After this is done, the SVM, Decision Tree, or PPR classification is performed on the unlabeled data, depending on the input from the user. From here, the false positive and miss rates are computed for each class on the newly labeled data. For this project, we used the following definition of False Positive Rate (FPR) from [10]:

$$FPR = \frac{FP}{(FP + TN)}$$

We also used the equation $MR = \frac{FN}{TP+FN}$to compute the Miss Rate (MR).

In the above equations:

- TP – The number of points correctly labeled for a given class; that is, an image is class *c* and was correctly labeled as *c*.
- TN – The number of points correctly not labeled as a given class; that is, an image is not class *c* and was correctly labeled as a different class from *c*.
- FP – The number of points not in class *c* incorrectly labeled as class *c*.
- FN – The number of points in class *c* which were not labeled as class *c*.

The statistics for FPR and MR were computed by iterating over all of the now-labeled images for each class. Additionally, an overall accuracy rate (the number of correctly labeled images compared to the total number) was computed. The classification results and statistics are then saved to a text file for later retrieval.

Output Format and Discussion of Results

The output for this set of tasks is twofold. First, if a user did not specify a latent semantic file to be used for a given task, then latent semantics are generated by the task. These are saved to disk for re-use in a binary pickle file. Essentially, this binary file stores a nested dictionary containing the image names as keys to dictionaries storing the various feature vectors (and original image data). This can be read in and re-used in Tasks 1-3 or other tasks which use a latent semantic file, such as Tasks 4, 5, and 8. The latent semantic file is saved to the user-provided output folder with the following, auto-generated filename:

```
<base_featuretype>_<method of feature decomposition>_<number of latent features>.p
```

In addition to the latent semantics generated, each task also generates an output file detailing the results of classification (again with an auto-generated title to the user-specified output folder). This file, with an example listed below (truncated for brevity), first lists the images provided for classification and then the labels assigned to each by the classifier. After this, the false positive and miss rates for each class, as well as a statistic providing the overall accuracy of class assignments.

```
The input images were classified as follows:
image-cc-1-6.png:           cc
image-cc-10-10.png:         cc
image-cc-19-4.png:          cc
image-cc-19-5.png:          cc
image-cc-20-1.png:          cc
image-cc-29-4.png:          cc
image-cc-3-5.png:           cc
image-cc-4-4.png:           cc
image-con-11-1.png:         con
image-con-11-2.png:         con

...

The classes had the following false positive rates:
cc:         0.0
con:        0.08602
emboss:     0.0
jitter:     0.02247
neg:        0.0
noise01:    0.04348
noise02:    0.08696
original:   0.07216
```

```
poster:     0.02151
rot:        0.0
smooth:     0.01087
stipple:    0.05102
The classes had the following miss rates:
cc:         0.0
con:        0.22222
emboss:     0.0
jitter:     0.92308
neg:        0.0
noise01:    1.0
noise02:    0.3
original:   0.6
poster:     0.77778
rot:        0.0
smooth:     0.0
stipple:    0.0
Overall accuracy: 0.6372549019607843
```

Our group's accuracy was typically best when using around 50 latent features extracted from ELBP using SVD, on Task 1. All models typically produced accuracy results of at least 40%, though this could often be higher (typically 50%-60% for all 3 classification algorithms). For Tasks 2-3, results were significantly worse. Although the results from the PPR classifier were typically somewhat better (in the 15%-40%) accuracy range, results for SVM and DT were far worse, hovering around 3%-10% at best.

Our group believes that, in general, better results were obtained with the 'default' parameters (k=50, model=ELBP or HOG, using SVD) on Task 1 for a number of reasons. First, we believe that classification results in Task 1 are typically better due to the fact that types of images are significantly more distinct between each other than subjects or IDs. This lends itself to easier classification, as it will be easier for all classification methods to find clear boundaries between types than other class labels. Additionally, better results were obtained with more latent features on ELBP with PPR and SVM, while better results were obtained with HOG on DT. This was somewhat surprising, but seemed to hold consistently across testing.

All 3 classification models worked with other inputs (k being 5, 10, or 20, for example) but produced significantly worse results due to the loss of data. Similarly, worse results were obtained on Tasks 2-3 due to the lack of distinguishing features between classes. Our implementations worked, but were simply unable to consistently distinguish between classes, so classification results were often suboptimal.

One final note is that our group's implementations of SVM and DT rely on some random choices. This has caused some instances where results on a given run of Tasks 1-3 may be significantly worse than expected. For example, our implementation of SVM occasionally labels all input images as being from the same class. Our group is not fully aware why this happens,

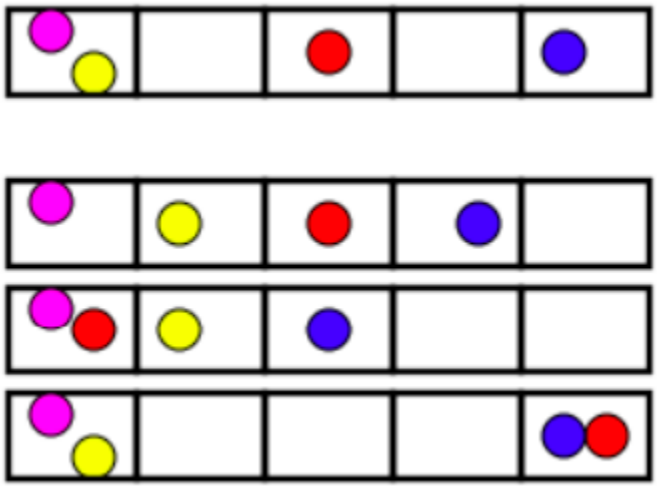
but if this occurs SVM and DT will often produce better results when run again with the same input parameters.

Tasks 4-5

LSH Index Structure

The objective of Task 4 was to implement a **LSH index structure**. The underlying logic behind LSH is to keep similar items in the same bucket with high probability of access when queried. This is a hashing technique that uses multiple hashing functions and encourages collisions. The method was initially discussed by Andoni and Indyk in [3], which discussed the underlying theory of LSH and various methods of computing the hashes for the index structure. How we have implemented this is as follows:

- We take the number of hashes and layers as input.
- A random vector is generated that has the dimensions of layers X hashes X input_len. This is used to multiply with the input array.
-
```
hashval="".join(['1' if i>0 else '0' for i in hash_value])
```
- The above snippet is used to create a binary hash for each image using values formed after the product. This effectively means that a ‘hash’ in a given layer is the sign of a dot product with a random vector. If the dot product is greater than 1, the vector hashes to 1, else the vector hashes to 0.
- This process is done for each layer and then stored in a final structure.



This is how the image points are stored in different buckets. The index structure is later used for future tasks.

```
{'01111111001111010110': ['test_images/image-cc-1-6.png']}
{'10101010011000001000': ['test_images/image-cc-1-6.png']}
{'10001010011101000100': ['test_images/image-cc-1-6.png']}
{'00101110010011010000': ['test_images/image-cc-1-6.png']}
{'10111000001100111111': ['test_images/image-cc-1-6.png']}
```

The second half of the task was to find the top t images for the query image. This query image also provided hash values as done above. Now we compared these values layer by layer to the images to find matches. If the matches found were less than t, we reduce the hash value and compare larger buckets.

For example, if the hash value was `11001` and not enough results were returned on the first pass, we then use only the first 4 bits to compare with the buckets present. This increases the search radius and gets us more matches. We then store the relevant images, filtering and only keeping unique images. These values are printed to the display.

LSH Output and Discussion of Results

The output of LSH is structured to show the total bytes, total buckets visited, overall images visited and unique images visited in the index structure. The top *t* similar images have been listed, after which we find the common images. These are used to find the false positive and miss rates from the top *t* images. Below is an example of the output printed to the terminal after running Task 4 with a number of input parameters:

```
python3 task4.py -l 5 -k 10 -t 10 -q test_images/image-cc-19-5.png -m elbp
-o Outputs -f phase3_data/2000
bytes: 22944
Total Buckets Visited: 364
Overall Images Visited: 364
Unique Images Visited: 323
10 similar images for the query image test_images/image-cc-19-5.png

ImageId: image-cc-19-8.png
ImageId: image-cc-19-10.png
ImageId: image-cc-19-2.png
ImageId: image-cc-16-2.png
ImageId: image-cc-7-5.png
ImageId: image-cc-30-7.png
ImageId: image-cc-8-5.png
ImageId: image-cc-30-1.png
ImageId: image-cc-7-1.png
ImageId: image-cc-27-4.png
Intersection of results and standard Euclidean distance:
['image-cc-19-10.png', 'image-cc-30-7.png', 'image-cc-7-5.png',
'image-cc-19-8.png', 'image-cc-16-2.png', 'image-cc-19-2.png']
The total misses from top t: 4
The total false-positives from top t: 4

The miss rate for the top t: 0.4
The false positive rate for the top t: 0.4

Including all images returned from the initial query (before filtering),
the false positive rate is: 0.9814241486068112

Results saved to 'Outputs/task4_img_results.png'
Saved LSH index file to 'Outputs/lsh_phase3_data-2000_5_10.p'!
Saved query results to Outputs/query_results_phase3_data-2000_4_4043.p!
This file can be used in Tasks 6-7.
Size in bytes of everything stored for LSH: 106571296
Execution time:13.199339151382446 seconds
```

An example image has been provided to show an example of the query results. Note that, in the results above, a brute-force Euclidean distance (where the distance from the input image to all images in the folder) is used as comparison. Euclidean distance is used here, as it provides the best results across all vectors (latent or not).

In testing, LSH provided consistently good query results, and often returned at least 60% of the closest images to a query point (when compared to standard brute-force Euclidean distance against all input images). This was often better, and 90%-100% of the correct *t* closest images were often returned with optimal parameters (both *k* and *L* as either 5 or 10). With the number of hashes per layer, and the number of layers, around these values, typically only 100-300 images were visited, as opposed to all images in the database. Increasing the number of hashes per layer reduced the number of visited images, but also reduced accuracy somewhat; similarly, increasing the number of layers much more than 10 often returned too many images in the initial search.

False positive and miss rates were computed against the results obtained from the brute force approach. These values were always the number of elements not in the intersection divided by the total, since the miss and false positive rate is always the same in this case. To see how accurate the hashing in LSH was, an additional false positive rate was computed from all images returned in the initial search, before filtering. This often hovered in the 90%-95% range, which was good.

The size of the LSH structure (which only includes the hash results) was typically 22000-23000 bytes, though this increases with the number of layers and hashes per layer. The amount saved to disk, including all computed feature vectors and a large amount of extraneous information, typically hovered in the range from 50MB - 100MB (note that this is a lot of data, much of it unnecessary for easier processing).

The output of Task 4 saves two output files to disk:

- An index file containing information about the LSH index structure and all image data. This allows all data to be retrieved. The format of this is:

```
['lsh', layers, hashes, Lsh, lsh_conversion_vector, image_dict, model,
folder_path, dim_red_technique, results]
```

- A saved query file containing the results of the query. This is used to read the query results in Tasks 6-7. It is of the following format:

```
[input_folder, query_image, query_image_name, processed_query_vector,
tasknum, query_results, t]
```

VA-Files Index Structure

A Vector Approximation File is used for similarity search in high dimensional spaces. It follows a filter-based approach of signature files instead of a conventional data-partitioning approach in which space is divided into cells. Vectors are then assigned approximations based on cells in which they lie. The VA-File list contains these small, bit-encoded approximations. In our project, we use the implementation of VA-Files by Blott and Weber in [4]; all images in this section and the overall algorithm is based on this paper.

The process starts from scanning all small approximations first, then using Nearest Neighbor queries to visit only a fraction of vectors (VA-File is used as a simple filter). Its performance is based on the degree 45 vector of color features. Even when the dimensionality increases, the performance of the VA-File algorithm doesn't degrade, instead it sometimes increases as the number of dimensions increases.

Instead of scanning large volumes of primary vector data, the VA-File has small approximations to each vector (10-15% of space of vectors). Based on these smaller approximations, many vectors can be excluded from the search, which makes the computation fast and cost effective. Queries are also points in the same space as vector data. Queries can be based on the Nearest Neighbor algorithm or its extensions like Partial Match, Weighted Search and Range Queries. The VA-File vector/ list has a simple flat structure which makes it easy for parallelization and straight-forward distribution. Weighted search, partial match and conjunctive queries are well supported.

Background

Similarity using VA-File is measured in terms of nearness using LP metrics. Dimensions are assumed to be independent of each other. Real time data in high dimensional space is often correlated and clustered & data occupy only a subspace. Hence, we can apply a variety of transformations like SVD or KL-Divergence.

Data Partitioning Index Methods

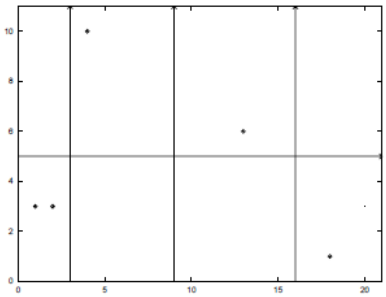
Space is partitioned into cells & a page is allocated to each cell. Data is partitioned & vectors are allocated to the pages corresponding to the cells. An index structure over these pages is built (e.g. Tree, dictionary). This index is used to prune search spaces during query processing. Metric distances (LP) can also be used for indexing. Indexes for metric distances typically partition data based on its proximity to one or more partition objects (as in Nearest Neighbour search). Then the data is partitioned and pruned for the search query.

Signature Methods

Signature is a small abstraction or approximation of an object. Efficient bitwise operations can be applied on signatures to prune objects which don't match the query. Signature file is a filtering technique. VA-Files result is similar to that of Signature methods, however the working of both differs greatly.

VA-File Approach

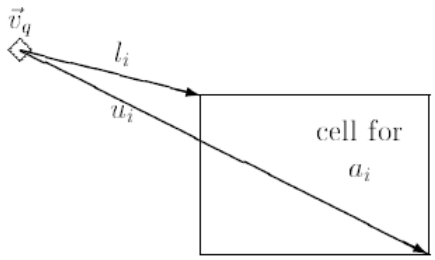
Vector space is partitioned into cells. Cells are used to generate bit encoded approximations for each vector. A VA-File is then created, which is a flat array of all the approximations. Query Processing is then done where all the approximations are scanned. Each approximation determines a lower and upper bound on the distance between the vector and the query. Bounds eliminate the vectors from the search, thus pruning the number of comparisons.



A b-bit approximation is calculated for each vector. First the bits b_j is calculated for each dimension j, which follows the following relation based on the number of bits (b) and number of dimensions (d)

$$b_j = \left\lfloor \frac{b}{d} \right\rfloor + \begin{cases} 1 & j \leq b \bmod d \\ 0 & otherwise \end{cases}$$

Then the lower bound and upper bounds are calculated for each query vector and distance functions are applied to each point to find the nearest neighbours. The below diagram, from [4], represents the upper bound and lower bound of a cell.



Lower and upper bounds for $L_p(\vec{v}_q, \vec{v}_i)$ given a_i guaranteeing $l_i \leq L_p(\vec{v}_q, \vec{v}_i) \leq u_i$

Now the search algorithm (VA-SSA : Vector Approximation Simple Search Algorithm) is applied to them to find k-most similar neighbours for each query object.

One crucial note is that, due to the method of query retrieval used in this project for VA Files (SSA, as described by Blott and Weber in [4]), the exact number of query results are returned by the initial query. This is problematic for query refinement, which relies on there being additional, extraneous query results (so that results can be iteratively improved upon). As such, if the user searches for *t* images, *5t* images are always returned from the query to the VA File index structure. This ensures that there are always enough images for query refinement in Tasks 6-8.

This problem could have likely been avoided if a different method of query retrieval was used, such as the "'Near-Optimal' Search Method" described in [4]. However, the project specification did not specify a specific search strategy to use on VA Files. As such, this concession in search results was made to ensure that Tasks 6-8 worked properly with query results obtained from VA Files.

VA-Files Output and Discussion of Results

The outputs of VA-Files are very similar to those from the LSH index files created in Task 4. Below is an example output from a run of Task 5:

```
python3 task5.py -b 3 -t 10 -q test_images/image-cc-1-6.png -d
Outputs/elbp_svd_20.p -o Outputs -x -m elbp
Size in bytes of the generated VA File: 14467.5
Total Buckets Visited: 20
```

```
Overall Images Visited: 153
10 similar images for the query image test_images/image-cc-1-6.png

ImageId: image-cc-35-8.png
ImageId: image-cc-31-1.png
ImageId: image-cc-38-4.png
ImageId: image-cc-37-8.png
ImageId: image-cc-38-10.png
ImageId: image-cc-24-4.png
ImageId: image-cc-6-4.png
ImageId: image-cc-6-5.png
ImageId: image-cc-26-6.png
ImageId: image-cc-13-8.png
Intersection of results and standard Euclidean distance:
['image-cc-24-4.png', 'image-cc-37-8.png', 'image-cc-38-4.png',
'image-cc-6-5.png', 'image-cc-13-8.png', 'image-cc-26-6.png',
'image-cc-35-8.png', 'image-cc-6-4.png', 'image-cc-38-10.png',
'image-cc-31-1.png']
The total misses from top t: 0
The total false-positives from top t: 0

The miss rate for the top t: 0.0
The false positive rate for the top t: 0.0

Including all images returned from the initial query (before filtering),
the false positive rate is: 0.934640522875817

Results saved to 'Outputs/task5_img_results.png'
elbp_svd_20
Saved VA index file to 'Outputs/va_elbp_svd_20_3.p'!
Saved query results to Outputs/query_results_elbp_svd_20_5_1371.p!
This file can be used in Tasks 6-7.
Size in bytes of everything stored for VA File: 106759104
Execution time:1.149350881576538 seconds
```

Similar to before, false positive and miss rates are computed on the data, as well as the number of buckets visited (in this case, the cells examined by the algorithm) and unique images returned. For all cases, due to using SSA to query the index structure (as described in [4]) there are no misses. For false positives, when comparing against all images returned, the best results were obtained with higher values of b, which increased the number of cells (and reduced the false positive rate). However, the algorithm ran faster and the index file used less space with choices of fewer bits per dimension; in testing, values of 3 or 5 provided good results with minimal time or space tradeoff.

One important note about the SSA search in VA-Files is that the false positive rate is somewhat inflated. Due to the need to have query results for user relevance feedback, the number of images returned is larger than necessary. This causes more images to be visited in the index structure than absolutely necessary, but does give a rough estimate of the percentage in general (for any choice of *t*). Below is an example output from a query run on the VA File structure.

Tasks 6-7

Overview

For Taks 6-7, we implemented programs which are able to take queries performed on an index structure generated in Tasks 4-5 (LSH or VA Files) and then refine these results. First, a user is presented with a visual representation of the top ‘t’ results returned by the query on the index structure. Below is an example of an image presented to a user summarizing the query results (in this case, image `image-cc-19-5.png` was compared to the images in the `1000` dataset using the LSH hash index structure).

After the user has a chance to view the results of the previous query, they are then able to select which images are relevant (and not) via command-line in the terminal. For this project, the `TerminalMenu` from `simple_term_menu` was used to accomplish this task. Once the user has selected some (but neither none nor all) images as relevant and irrelevant, the data is then passed to the Decision Tree or SVM classifiers for further refinement (in Tasks 6 and 7 respectively).

```
➜python3 task7.py -i Outputs/lsh_phase3_data-1000_5_10.p -q Outputs/query_results_phase3_data-1000_4_8588.p -o Outputs
LSH index file detected!
This index file was created with the following parameters:
        Input Folder:        phase3_data/1000
        Number of Layers:    5
        Hashes per Layer:    10
        Feature vector used: latent
Results saved to 'Outputs/task7_temp_img.png'
> [ ] image-cc-19-2.png
  [ ] image-cc-25-3.png
  [ ] image-cc-36-3.png
  [ ] image-cc-30-1.png
  [ ] image-cc-12-9.png
  [ ] image-cc-9-9.png
  [ ] image-cc-25-10.png
  [ ] image-cc-23-8.png
  [ ] image-cc-9-2.png
  [ ] image-cc-5-10.png
Press <space>, <tab> for multi-selection and <enter> to select and accept
```

Once revised query results are presented to the user (in an image format similar to the example above), the user can choose to quit or iterate on the query refinement. If the user chooses to refine the query further, they are presented with another selection menu which augments the selections made in earlier iterations. The combined results are then used to form a new

classification and revision of query results. This continues until all images are listed as ‘relevant’ or ‘irrelevant’, or the user chooses to end execution.

A crucial note about the query revising done in these tasks is that this is effectively re-ordering the results provided in the *initial* query to the index structure. Thus, the query refinement is not especially effective unless there are more images returned by the query than necessary (for example, if the top 10 images were desired, the search through the index returned 50 or more). While this means that very effective query refinement can be achieved when the initial query is not especially accurate, the refinement is not possible or very useful when the initial query provides very tight results.

Query Refinement with Decision Tree

Once the images that are relevant are selected from the query output of task 4 or 5, we label these results as “1” to denote it as relevant. The unselected query results are labeled as “0” to mark them as irrelevant for the user.

After labelling, we use the decision tree to build the classifier with this data as input. Now we use this tree that contains information captured about user feedback to label the other unlabeled images. The result of this prediction along with the list of initially selected images by the user gives us all the relevant images to consider.

We finally compute the Euclidean distance metric between every selected image with the query image and then sort these results in increasing order of distance to then finally return the new “t” set of top results.

The results when using Decision Tree for query refinement were similar to the results encountered when using SVM, as described below. Decision Tree often provides better results in fewer iterations, however. Below is an example of query results before, and after, multiple rounds of query refinement (note that some suboptimal choices were made by the user in this example).

Top 20 query results for query image 'test_images/image-con-18-8.png':

Query Image Filename: test_images/image-con-18-8.png

Image Name: image-con-18-3.png

Image Name: image-con-18-5.png

Image Name: image-con-18-10.png

Image Name: image-con-15-2.png

Image Name: image-con-18-7.png

Image Name: image-con-18-6.png

Image Name: image-con-15-9.png

Image Name: image-con-18-9.png

Image Name: image-poster-10-8.png

Image Name: image-stipple-1-1.png

Image Name: image-con-15-7.png

Image Name: image-stipple-1-3.png

Image Name: image-con-3-9.png

Image Name: image-con-9-8.png

Image Name: image-poster-17-8.png

Image Name: image-poster-16-7.png

Image Name: image-con-38-6.png

Image Name: image-con-1-1.png

Image Name: image-poster-1-1.png

Image Name: image-con-1-3.png

Top 16 query results for query image 'test_images/image-con-18-8.png':

Query Image Filename: test_images/image-con-18-8.png

Image Name: image-con-18-3.png

Image Name: image-con-18-10.png

Image Name: image-con-18-7.png

Image Name: image-con-15-9.png

Image Name: image-stipple-1-1.png

Image Name: image-stipple-1-3.png

Image Name: image-con-16-3.png

Image Name: image-con-40-10.png

Image Name: image-con-12-9.png

Image Name: image-con-20-8.png

Image Name: image-stipple-12-1.png

Image Name: image-con-1-7.png

Image Name: image-con-16-2.png

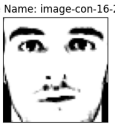


Image Name: image-con-20-9.png

Image Name: image-con-3-10.png

Image Name: image-con-40-2.png

Query Refinement with SVM

To refine queries with SVM, our group used a method similar to that discussed by Drucker, H., Shahrary, B., & Gibbon in [8], who discuss how to use user feedback of image relevance to revise queries. Essentially, the approach uses the user feedback as an initial labeled dataset for classification for SVM (where a hyperplane is found which divides the labeled data). Then, the remaining images are assigned a classification and 'relevant' images are re-ordered for new query results. The below image is a figure from [8] which illustrates the desired approach to find a single line to separate the two groups of points. It is important to note that the approach taken here only reorders the rankings of points from the initial query on the index; new data from the entire dataset is not considered.

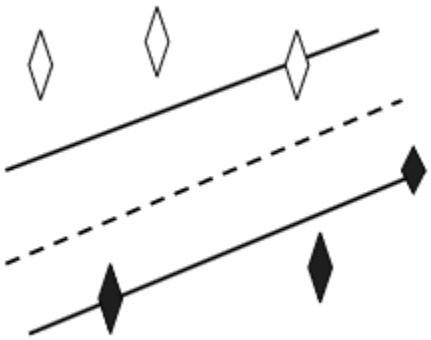


Fig. 1. Support vectors and separating hyperplanes.

Our implementation is very similar. First, the user marks the *t* images returned from the initial query as 'relevant' or 'irrelevant'. This initial set of data is treated as our initial labeled data for training (with labels of '1' or '0', respectively), and is used to train a specialized, basic version of SVM only able to perform binary classification on data. Once the SVM model is trained, the remaining data returned from the initial query is run through the model and assigned a class label of 'relevant' or 'irrelevant'. From the images labeled 'relevant' (including the points the user initially labeled) a new set of *t* ranked query results are selected, ordered based on the distance from the separating hyperplane, with points further from the hyperplane being ranked more highly, similar to the method of ranking discussed in [8].

These new query results are then presented to the user in image format. The user can then choose to further refine the image results from the results presented to them, again labeling images as 'relevant' or 'irrelevant'. This will re-order the query results again, and can be done indefinitely until too many images are labeled by the user or the user chooses to stop.

In testing, it was found that at least 2 rounds of query refinement was typically required to produce sufficiently accurate results. The below sequence of 3 images shows the progression of query refinement over 2 iterations, where the user was selecting images of subject 19 and 'cc' images without glasses as 'relevant'. Notably, the first set of results are somewhat inaccurate,

but the second set of results is much better. This is likely due to the fact that multiple iterations allow enough training data to be generated by the user such that the classification can be accurate.

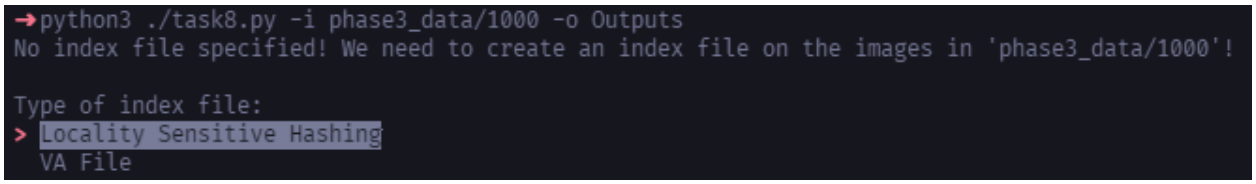
One important caveat of the SVM query refinement method implemented in this project is that it is exceptionally slow when run on long feature vectors. For the smallest base feature vectors,

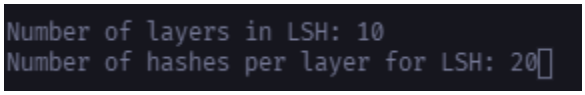
color moment, which are 640 elements long, the algorithm takes 8-10 minutes to reclassify images based on user feedback. This is clearly not optimal. To avoid this, only latent feature vectors (of length of around 100 at most) should be used with this method of query refinement to ensure that the runtime does not grow too long.

Task 8

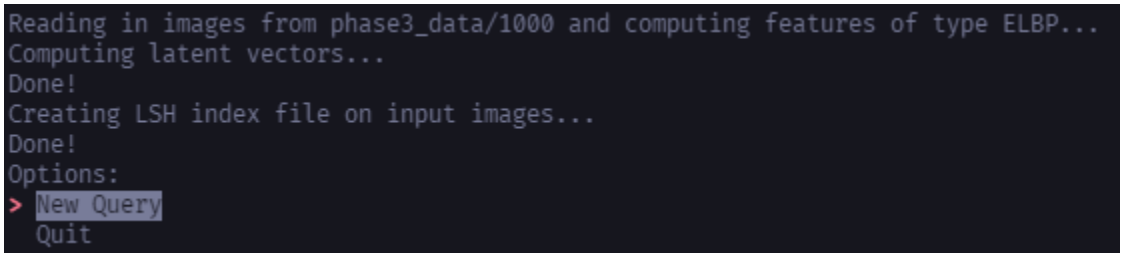
For Task 8, we combined the work performed in Tasks 4-7 into one interactive program which could be entirely controlled from the command line. This allows a user to create/load an index file then perform queries on that index structure. The queries can be refined further using SVM or Decision Tree classifiers on the results returned by the initial query in the same method that query results can be iteratively improved on in Tasks 6-7.

To allow the user to interact with the program, most options are selected from a menu in the terminal, generated with the `TerminalMenu` from `simple_term_menu`. From here, a user is able to select various options, including what parameters to generate the initial feature vectors with (if a latent semantic file was not specified). One example menu is shown below; other menu options are very similar.





After an index file is created on the images in the input folder, the user is given an option to create a new query on the index file. After an input image is specified and a number of images to return in the query is provided, the program finds the query results with methods identical to those used in Tasks 4-5, for LSH and VA index files respectively. After refining the query as much as desired, the user is able to return to this point and perform a new query on the data, if desired.



After performing the initial query on the data, the user is then able to view the query results in an image format (which are saved to an image file in an output folder). From these results, the user can then decide whether or not to refine results using SVM or Decision Tree classifiers. This

process, including menu options and output format, is nearly identical to the approach discussed in the section for Tasks 6-7, and will not be included here for brevity.

Essentially, the program in Task 8 uses a series of terminal menus and common library functions to combine multiple tasks into one, effectively allowing a user to perform all the functionality in Tasks 4-7 — create an index file, perform a query on an index file, and refine a query with user input — from one program, without the need to save intermediate data or remember filenames. However, this approach means that many of the same problems from the aforementioned tasks, especially the difficulty in refining query results when few results were returned from the initial query, remain.

Interface Specification

All tasks in the project can be run from the command line. All instructions below are given with the assumption that the user is running a Linux-based operating system with the appropriate Python modules installed; the instructions should hold true for other operating systems (but were not tested). Each command will be given in the format of the command with the flags used from the command line. Explanations of the flags are provided below each command.

Task 1

```
python3 task1.py -i <INPUT_FOLDER> -m <MODEL> -d <DIM_RED_TECHNIQUE> -k <K>
-o <OUTPUT_FOLDER> -l <LATENT_SEMANTIC_FILE> -f <CLASSIFY_FOLDER>
-c <CLASSIFIER>
```

- `-i` – The input folder of images used for the task. This set of images is processed and used as the training data for classification. This is an optional parameter, and is not required if a latent semantic file is given.
- `-m` – The color model used to extract features from the images. This should be '`color`', '`elbp`', or '`hog`' to use the color moment, ELBP, and HOG feature descriptors respectively. This is an optional parameter, and is not required if a latent semantic file is given.
- `-d` – The method used to extract the latent semantics from the dataset. This should be one of the strings '`pca`' or '`svd`' to use the PCA or SVD dimensionality reduction methods, respectively. This is an optional parameter, and is not required if a latent semantic file is given.
- `-k` – The number of latent semantics to select (where the top k are selected). This value should be an integer that is at least 1 and smaller than the original length of the feature vector used. Any choice in the range 5 to 100 is reasonable, though other choices are possible. A value of '*' can be specified to allow the user to use base feature vectors, as opposed to latent features; this is not recommended. This is an optional parameter, and is not required if a latent semantic file is given.
- `-o` – The output folder to which results are saved. Note that multiple files are created and named automatically by the program.
- `-l` – A saved latent semantic file used to avoid re-computation of data. This is an optional parameter, but using this means that one does not have to specify the optional parameters used to extract latent features.
- `-f` – The folder of images which are to be classified by the program.
- `-c` – The classifier to be used to classify the different images. This should be one of the strings '`svm`', '`dt`', or '`ppr`' to use an SVM, Decision Tree, or PPR classifier, respectively.

Task 2

```
python3 task2.py -i <INPUT_FOLDER> -m <MODEL> -d <DIM_RED_TECHNIQUE> -k <K>
-o <OUTPUT_FOLDER> -l <LATENT_SEMANTIC_FILE> -f <CLASSIFY_FOLDER>
-c <CLASSIFIER>
```

- `-i` – The input folder of images used for the task. This set of images is processed and used as the training data for classification. This is an optional parameter, and is not required if a latent semantic file is given.
- `-m` – The color model used to extract features from the images. This should be '`color`', '`elbp`', or '`hog`' to use the color moment, ELBP, and HOG feature descriptors respectively. This is an optional parameter, and is not required if a latent semantic file is given.
- `-d` – The method used to extract the latent semantics from the dataset. This should be one of the strings '`pca`' or '`svd`' to use the PCA or SVD dimensionality reduction methods, respectively. This is an optional parameter, and is not required if a latent semantic file is given.
- `-k` – The number of latent semantics to select (where the top k are selected). This value should be an integer that is at least 1 and smaller than the original length of the feature vector used. Any choice in the range 5 to 100 is reasonable, though other choices are possible. A value of '*' can be specified to allow the user to use base feature vectors, as opposed to latent features; this is not recommended. This is an optional parameter, and is not required if a latent semantic file is given.
- `-o` – The output folder to which results are saved. Note that multiple files are created and named automatically by the program.
- `-l` – A saved latent semantic file used to avoid re-computation of data. This is an optional parameter, but using this means that one does not have to specify the optional parameters used to extract latent features.
- `-f` – The folder of images which are to be classified by the program.
- `-c` – The classifier to be used to classify the different images. This should be one of the strings '`svm`', '`dt`', or '`ppr`' to use an SVM, Decision Tree, or PPR classifier, respectively.

Task 3

```
python3 task3.py -i <INPUT_FOLDER> -m <MODEL> -d <DIM_RED_TECHNIQUE> -k <K>
-o <OUTPUT_FOLDER> -l <LATENT_SEMANTIC_FILE> -f <CLASSIFY_FOLDER>
-c <CLASSIFIER>
```

- `-i` – The input folder of images used for the task. This set of images is processed and used as the training data for classification. This is an optional parameter, and is not required if a latent semantic file is given.
- `-m` – The color model used to extract features from the images. This should be '`color`', '`elbp`', or '`hog`' to use the color moment, ELBP, and HOG feature descriptors respectively. This is an optional parameter, and is not required if a latent semantic file is given.
- `-d` – The method used to extract the latent semantics from the dataset. This should be one of the strings '`pca`' or '`svd`' to use the PCA or SVD dimensionality reduction methods, respectively. This is an optional parameter, and is not required if a latent semantic file is given.
- `-k` – The number of latent semantics to select (where the top k are selected). This value should be an integer that is at least 1 and smaller than the original length of the feature vector used. Any choice in the range 5 to 100 is reasonable, though other choices are possible. A value of '*' can be specified to allow the user to use base feature vectors, as opposed to latent features; this is not recommended. This is an optional parameter, and is not required if a latent semantic file is given.
- `-o` – The output folder to which results are saved. Note that multiple files are created and named automatically by the program.
- `-l` – A saved latent semantic file used to avoid re-computation of data. This is an optional parameter, but using this means that one does not have to specify the optional parameters used to extract latent features.
- `-f` – The folder of images which are to be classified by the program.
- `-c` – The classifier to be used to classify the different images. This should be one of the strings '`svm`', '`dt`', or '`ppr`' to use an SVM, Decision Tree, or PPR classifier, respectively.

Task 4

```
python3 task4.py -l <NUM_LAYERS> -k <NUM_HASHES_PER_LAYER>
-t <NUM_QUERY_RESULTS> -q <QUERY_IMAGE> -m <MODEL> -o <OUTPUT_FOLDER>
-f <INPUT_DATA_FOLDER> -d <LATENT_FILE> -x
```

- `-l` – The number of layers to use with LSH in this task. This should be a positive integer.
- `-k` – The number of hashes per layer of the LSH data structure. This should be a positive integer.
- `-t` – The number of images to return from a query. This should be a positive integer.
- `-q` – The query image used for the query against the created LSH structure.
- `-m` – The color model used to extract features from the images. This should be '`color`', '`elbp`', or '`hog`' to use the color moment, ELBP, and HOG feature descriptors respectively.
- `-o` – The output folder to which results are saved. Note that multiple files are created and named automatically by the program.
- `-f` – The folder of images which are to be stored in the LSH index file. This is an optional parameter and not required if the latent file is specified.
- `-d` – A saved latent semantic file used to avoid re-computation of data. This is an optional parameter, but using this means that one does not have to specify the optional parameters used to extract latent features.
- `-x` – An optional flag with no input associated. If set, uses latent features instead of the 'base' feature vector.

Task 5

```
python3 task5.py -b <BITS_PER_DIMENSION> -t <NUM_QUERY_RESULTS>
-q <QUERY_IMAGE> -m <MODEL> -o <OUTPUT_FOLDER> -f <INPUT_DATA_FOLDER>
-d <LATENT_FILE> -x
```

- `-b` – The number of bits to use per dimension in the VA index structure. This should be a small, positive integer.
- `-t` – The number of images to return from a query. This should be a positive integer.
- `-q` – The query image used for the query against the created LSH structure.
- `-m` – The color model used to extract features from the images. This should be '`color`', '`elbp`', or '`hog`' to use the color moment, ELBP, and HOG feature descriptors respectively.
- `-o` – The output folder to which results are saved. Note that multiple files are created and named automatically by the program.
- `-f` – The folder of images which are to be stored in the LSH index file. This is an optional parameter and not required if the latent file is specified.

- `-d` – A saved latent semantic file used to avoid re-computation of data. This is an optional parameter, but using this means that one does not have to specify the optional parameters used to extract latent features.
- `-x` – An optional flag with no input associated. If set, uses latent features instead of the 'base' feature vector.

Task 6

```
python3 task6.py -i <INPUT_INDEX_FILE> -q <INPUT_QUERY_RESULTS>
-o <OUTPUT_FOLDER>
```

- `-i` – An index file to use as input. Should be an LSH or VA file generated in Tasks 4-5.
- `-q` – A saved set of query results to use as input. Should be generated from a query in Tasks 4-5.
- `-o` – The output folder to which results should be saved.

Task 7

```
python3 task7.py -i <INPUT_INDEX_FILE> -q <INPUT_QUERY_RESULTS>
-o <OUTPUT_FOLDER>
```

- `-i` – An index file to use as input. Should be an LSH or VA file generated in Tasks 4-5.
- `-q` – A saved set of query results to use as input. Should be generated from a query in Tasks 4-5.
- `-o` – The output folder to which results should be saved.

Task 8

```
python3 task8.py -i <INPUT_FOLDER> -f <INDEX_FILE> -o <OUTPUT_FOLDER>
```

- `-i` – The input folder from which images should be read. An optional parameter which should be used only if an index file is not specified.
- `-f` – An index file saved to disk. This is an optional parameter, and should not be specified if an input folder is provided.
- `-o` – The output folder to which results should be saved.

Miscellaneous Files

A small number of helper files have been included to help generate files and print binary files to the terminal. Instructions to use the interfaces for these programs has been included, in case a reader wishes to use these functions.

`create_latent_file.py`

This file allows a user to create a latent semantic file and store it, if necessary. This avoids the need to use Tasks 1-3 to generate and save a latent semantic file.

```
python3 create_latent_file.py -i <INPUT_FOLDER> -m <MODEL>
-d <DIM_RED_TECHNIQUE> -k <K> -o <OUTPUT_FOLDER>
```

- `-i` – The input folder of images to generate the latent semantics on.
- `-m` – The color model used to extract features from the images. This should be '`color`', '`elbp`', or '`hog`' to use the color moment, ELBP, and HOG feature descriptors respectively.
- `-d` – The dimensionality reduction technique to use to extract features. Should be '`pca`' or '`svd`' for PCA and SVD respectively.
- `-k` – The number of latent semantics to generate.
- `-o` – The output folder to save the results.

`print_index_file.py`

This file allows a user to create a latent semantic file and store it, if necessary. This avoids the need to use Tasks 1-3 to generate and save a latent semantic file.

```
python3 print_index_file.py -i <INPUT_INDEX_FILE>
```

- `-i` – The input index file to read and print to the terminal.

System Requirements and Installation Instructions

The project was created in Python 3.8 and run/tested on a Ubuntu 20.04 system. In order to run this project, a version of Python 3.8+ is required (due to some new Python features). Any system that runs Python, such as Windows, MacOS, or other distributions of Linux, will likely be suitable to run the project, but this is not guaranteed. The exception to this is that Tasks 6-8 will likely not run on Windows systems, as the method used to create the user interface (the `TerminalMenu` from `simple_term_menu`) does not work on Windows systems.

In addition to a modern version of Python and an up-to-date operating system, there are a number of non-standard Python libraries which are required to run the project. These can be installed with `pip3` (or an alternate Python library installer):

- `numpy`
- `pickle`
- `argparse`
- `skimage`
- `simple_term_menu`
- `os`
- `sklearn`
- `pandas`
- `itertools`
- `scipy`
- `matplotlib`
- `progressbar2`

As long as the required libraries are installed, there is no specific installation required for the project. To run the Tasks for the project, there are a number of Python files (`task1.py` through `task8.py`) in the Code folder provided. In order to run these files, they must be in the same directory as the provided `lib` folder, which contains a number of custom Python functions necessary to run the Task files. As long as this condition is met, all files will be able to be run successfully. To view instructions on how to actually run the Task files, please refer to the above section on the interface specification.

Related Work

In [11], Lin et al. discuss methods of using SVM for image classification on extremely large datasets of size 4GB-8GB. In addition to needing to address the problem of effectively classifying the images, the paper discusses the approaches required to quickly extract features from the dataset, since the number of images is so large. To do this, the authors in [11] use coordinate coding and parallelization to perform feature extraction in a timely manner. Once features are extracted, they discuss how to use an Averaging Stochastic Gradient Descent (ASGD) method to compute SVM classification results, as well as how to perform this in parallel for efficient computation. The results and techniques used in this paper could be useful to consider when thinking of methods to improve the speed of SVM classification, especially on large datasets. Paper [15] discussed image classification using Artificial Neural Net (ANN) and Support Vector Machine where a two-layer classifier is used for Adult Image Classification.

Huang, Weng, Liu, and He discuss a new type of index structure, an HD-Tree, in [12]. This tree is an extension of an existing type of storage index, the D-Tree, which (simplifying greatly) is a quadtree that distributes data by splitting the original space of objects into multiple subspaces [12]. The HD-Tree expands upon this by using a “half-decomposition” [12] strategy to save space and improve time performance. The paper details the algorithm for the tree itself and shows that the HD-Tree is an improvement, both in insertion and query performance, over D-Trees and other index structures. Although the index structure shown in [12] is very complex and beyond the scope of this class, it is interesting to consider the improvements which could be made if such a structure were used, if any. Perhaps using an HD-Tree, as opposed to LSH or VA-Files, could produce better storage and query results for the data in this project..

One important problem considered during this phase of the project was query relevance feedback from a user, and how to best use the feedback that an image is ‘relevant’ or ‘irrelevant’ in re-ordering query results. Muller et al. consider a number of ways to accomplish this task in [13], as well as a number of strategies to improve query results after accepting user feedback. In particular, the paper discusses several methods of doing automated feedback. This includes using only positive feedback, positive and negative feedback, various weightings for feedback, and several steps/iterations of feedback [13]. Each example showed a graphical representation of how tweaking various values can affect the precision and recall of results. Ultimately, it was concluded that relevance feedback could enhance the query, but “too much negative feedback can destroy the query”, and considering other approaches (such as weighting) can improve results [13]. The techniques covered in this paper could be useful when considering other approaches to relevance feedback.

Although a number of image classification methods were used in this project (SVM, DT, and PPR), one crucial classification method that is often used was not considered: neural networks. In [14], Giacinto and Roli discuss how to efficiently create a neural network to correctly classify images, similar to the classification done in this project. Specifically, [14] discusses how to create neural networks with enough independent error in the neural networks to produce consistently correct results. This covers how to implement the neural network, as well as a summary of results. In future iterations of the project, it could be useful to consider neural networks, such as the implementation discussed in [14], to classify images.

Conclusions

In this project we have created classification engines which help users to identify types, subject or sample ID of images. We have used supervised classification techniques SVM and Decision Tree, and Personalized Pagerank based classifiers to achieve the goal. All the tasks were designed based on the concepts taught in the class. Specifically, this project gave us the opportunity to learn about supervised classifiers, Locality Sensitive Hashing and Vector Approximation files.

SVM was implemented with the assumption that objects are linearly separable, but we can only get decent accuracy, precision and recall in Task1 where we classified the image types. But for

other cases it became very difficult to find hyper planes which separate data according to their labels. Though we have tried with the polynomial and Radial Basis Function kernel, we were not able to improve the accuracy, precision and recall of Task 2 and Task 3 where we need to associate Subject ID and Sample ID respectively.

In addition to learning about image classification, this project was a useful experience to learn about how to store and retrieve feature vectors from index files, as well as perform/refine queries on this index. Our group ultimately received good results when doing this, but the work here could likely be improved upon as well through the use of better retrieval techniques and algorithms. Tying everything together in Task 8 was also a useful experience to see how many parts of multiple phases of the project fit together.

Ultimately, this phase of the project was a useful learning experience for the group and provided a useful baseline for how to correctly process images, extract/use latent semantics, perform indexing, classify the images and build classification based relevance feedback techniques. This was a useful exercise in applying the concepts learned in the course, and likely will be relevant if any members of the group are required to perform multimedia data processing in the future.

Bibliography

[1] Huang, S., Li, X., Candan, K. S., Sapino, M. L. (2016). Reducing seed noise in personalized PageRank. Social Network Analysis and Mining, 6(1), 1-25.
[2] F. Lin and W. W. Cohen, "Semi-Supervised Classification of Network Data Using Very Few Labels," *2010 International Conference on Advances in Social Networks Analysis and Mining*, 2010, pp. 192-199, doi: 10.1109/ASONAM.2010.19.
[3] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06), 2006, pp. 459-468, doi: 10.1109/FOCS.2006.49.
[4] Blott, Stephen & Weber, Roger. (1998). A Simple Vector-Approximation File for Similarity Search in High-Dimensional Vector Spaces.
[5] Weber, R., Schek, H., & Blott, S. (1998). A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB*.
[6] Candan Kasim Selçuk, & Sapino, M. L. (2010). *Data Management for Multimedia Retrieval*. Cambridge University Press.
[7] MacArthur, Brodley and Chi-Ren Shyu, "Relevance feedback decision trees in content-based image retrieval," *2000 Proceedings Workshop on Content-based Access of Image and Video Libraries*, 2000, pp. 68-72, doi: 10.1109/IVL.2000.853842.
[8] Drucker, H., Shahrary, B., & Gibbon, D. C. (2002). Support vector machines: relevance feedback and information retrieval. *Information processing & management*, *38*(3), 305-323.
[9] Olivetti Faces dataset, AT&T Laboratories Cambridge, 2001.
[10] Pico Quantitative Trading. (2021). *What is a false positive rate?* Pico. Retrieved November 28, 2021, from https://www.pico.net/kb/what-is-a-false-positive-rate/.
[11] Y. Lin *et al*., "Large-scale image classification: Fast feature extraction and SVM training," *CVPR 2011*, 2011, pp. 1689-1696, doi: 10.1109/CVPR.2011.5995477.
[12] Huang, T., Weng, Z., Liu, G., & He, Z. (2020). HD-tree: An efficient high-dimensional virtual index structure using a half decomposition strategy. *Algorithms*, *13*(12), 338. https://doi.org/10.3390/a13120338
[13] H. Muller, W. Muller, S. Marchand-Maillet, T. Pun and D. M. Squire, "Strategies for positive and negative relevance feedback in image retrieval," *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, 2000, pp. 1043-1046 vol.1, doi: 10.1109/ICPR.2000.905650.
[14] Giacinto, G., & Roli, F. (2001). Design of effective neural network ensembles for image classification purposes. *Image and Vision Computing*, *19*(9-10), 699–707. https://doi.org/10.1016/s0262-8856(01)00045-2
[15] Le, Hoang & Le, Thai & Tran, Son & Tran, Hai & Thuy, Nguyen. (2012). Image Classification using Support Vector Machine and Artificial Neural Network. International Journal of Information Technology and Computer Science. 4. 10.5815/ijitcs.2012.05.05.

Appendix

Split of Group Work

Below lists the main roles/work of each member:

- **Ayush Anand** – Implemented Locality Sensitive Hashing Tool for Task 4 and performed unit testing.
- **Pritam De** – Implemented Multiclass Support Vector Machine for Tasks 1, 2, and 3, implemented a binary SVM for Task 7, and performed unit testing.
- **Sairaj Menon** – Implemented Multiclass Decision Tree for Tasks 1, 2, 3, and 6 and performed unit testing.
- **Sritej Reddy** – Implemented VA-file based index structure and code to get the *t* most similar images from the above index-structure and other parts related to Task 5.
- **Aaron Steele** – Implemented the PPR image classification algorithm. Also handled most of the unifying of parts in the project in Task 8. Additionally, handled structure, I/O, and general flow of all tasks in the project, and performed integration work and testing.
- **Shubham Verma** – Read and Implemented VA-file based index structure to divide space into cells for approximate indexing and to find the most similar *t* Nearest Neighbours in Task 5.