CSE 515 Multimedia and Web Databases Phase 3

Clustering, Indexing and Classification

Project Report

(Group – 10)

Group Members

Devanshu Gupta

Sonal Sujit Prabhu

Surya Makkena

Vaibhav Somani

Vikas Kamineni

Vishal Reddy Eda

Abstract

In the phase 3 of the project, we explored, analyzed, and implemented clustering, indexing and classification mechanisms. We executed tasks to implement multidimensional scaling, DBScan, KNN classifier, decision-tree classifier, PPR based classifier, Locality Sensitive Hashing, SVM based classification.The phase 3 of the project consists of 6 tasks numbered 0 to 5 covering the above mentioned concepts. We used different python libraries to complete this project. The libraries that we used in the project are PIL, Pytorch, NumPy, pandas, sciPy, sklearn, Pickle, Matplotlib etc. The goal of this project is to familiarize ourselves with different similarity measures, dimensionality curse, graph representation. This project still uses the caltech 101 dataset present in the torchvision library. For this phase, we used the even numbered images as train data whereas the odd numbered images in the caltech dataset can be considered as test data. So, the total number of images used for training is 4339. In this project, the tasks 0a and 0b calculate the inherent dimensionality of the images using the layer_3 vector space extracted and stored in project phase 2. In Task1, we implement a SVD algorithm on the avgpool vector space. This task is similar to the

dimensionality reduction task we did in phase 2. The task 1 also involves predicting the most likely labels using the distance/ similarity functions on the reduced vector space. In Task 2, for each label in the dataset, we compute the c most significant clusters and these clusters are visualized using both differently colored point clouds in two dimensional Multidimensional Scaling space, and as groups of image thumbnails. In this task, we take a test image from the dataset and predict the most likely labels using the c label-specific clusters. In task 3, we implement different classifiers such as KNN classifier, decision tree classifier, Personalized Pagerank based classifier. For Tasks 1, 2, 3 we display the per-label precision, recall, F1-score and overall accuracy for each technique. In task 4, we implemented Locality Sensitive Hashing (LSH) for euclidean distance on the layer_3 vector space and stored the resultant index structure class in a pickle file. In task 4, we have a subtask 4b to visualize similar images using the index structure stored in 4a. The final task, task 5 is to implement a SVM based relevance feedback system and a Probabilistic relevance feedback system which considers the feedback provided by the user while retrieving the results.The project deliverables include well commented-code, Readme file, outputs, database files and a detailed report.

Keywords: Caltech101, Image Feature Extraction, Vectors, Vector spaces, Color Moments, Histogram of Oriented Gradients, ResNet50 Neural Architecture, ResNet-AvgPool-1024, ResNet-Layer3-1024, ResNet-FC-1000, distance/similarity measures, Classification, Latent Semantics, SVD, Multidimensional Scaling (MDS), Intrinsic or inherent dimensionality, dimensionality reduction, Precision, recall, F1-score, accuracy, DBScan, m-NN classifier, decision tree classifier, Personalized Pagerank based classifier (PPR), Locality Sensitive Hashing (LSH), Support Vector Machines (SVM) relevance feedback systems, Probabilistic Relevance Feedback System.

Introduction

The project has a total of 6 tasks covering concepts of Clustering, Indexing, Classification, and relevance feedback systems.

Terminology:

Caltech 101 Dataset: caltech 101 is the dataset present in the torchvision library which is used for this phase of the project.

Train Data: All the even numbered images in the caltech 101 dataset are used as training data.

Test Data: All the odd numbered images in the caltech 101 dataset are used as test data.

ResNet-50: ResNet-50 is a pretrained deep residual neural network layer composed of 50 layers which we used for feature extraction. The features are extracted from Avg_pool, Layer3 and Fully Connected layers of ResNet50.

Precision: Precision measures the effect of false hits. Precision is the ratio of true positive predictions to the total number of positive predictions made by the classifier. It is a measure of the accuracy of the positive predictions.

$$Precision = \frac{Retrieved\ and\ Relevant}{Retrieved}$$

Recall: Recall measures the effect of misses. It is the ratio of true positive predictions to the total number of actual positive instances. It is a measure of the classifier's ability to capture all positive instances.

$$Recall = \frac{Retrieved\ and\ Relevant}{Relevant}$$

F1 Score: F1 score is the harmonic mean of precision and recall. It provides a balance between precision and recall.

$$F1\ Score = \frac{2*(Precision * Recall)}{(Precision + Recall)}$$

Overall Accuracy: Overall accuracy is the ratio of correct predictions (both true positives and true negatives) to the total number of instances.

$$Overall\ Accuracy = \frac{(Retrieved\ and\ Relevant + Retrieved\ and\ not\ Relevant)}{(Retrieved\ and\ Relevant + Retrieved\ and\ not\ Relevant + Not\ Retrieved\ and\ Relevant + Not\ Retrieved\ and\ Not\ Relevant)}$$

Gini Impurity: Gini Impurity is a number between 0-0.5, which indicates the likelihood of new, random data being misclassified if it were given a random class label according to the class distribution in the dataset.

$$Gini\ Impurity = 1 - \sum_{i=1}^{k} p_i^2$$ (k is the number of labels)

Task 0:

- **Inherent Dimensionality or Intrinsic Dimensionality:** The minimum number of dimensions required to map higher level data into lower dimensions effectively without losing much information is known as Intrinsic Dimensions or Inherent Dimensionality. In simple words, the essential features that best represent/ characterize the dataset is known as Intrinsic dimensionality.
- **Multi Dimensionality Scaling (MDS):** Multi Dimensionality Scaling (MDS) is used to visualize the similarity or dissimilarity between the data. The goal of MDS is to preserve the distance or dissimilarity between objects in a dataset by representing

them in a lower dimensional space effectively. It works on the principle of minimizing stress.

- **Stress:** The stress defined here is used in the context of Multi Dimensionality Scaling. The stress is calculated as the square root of the difference between the sum of distances in the lower dimensions to that of the original data whole square divided by the sum of squares of distances in the original data. The formula for this equation is given below:

$$Stress = \sqrt{\frac{\sum_{i,j} \left(d^{\prime}_{ij} - d_{ij}\right)^2}{\sum_{i,j} {d_{ij}}^2}}$$

- **Label:** Each image in the dataset is mapped to a label which the image resembles to the most. This is something provided by the caltech101 dataset itself.

Task 1:

- **Label:** Each image in the dataset is mapped to a label which the image resembles the most. This is something provided by the caltech101 dataset itself.
- **EigenVectors and Values:** An eigenvector of a matrix is a non-zero vector which when multiplied by the matrix itself, returns a scalar multiple of the eigenvectors. Here, the scalar value is known as eigenvalue.
- **Dimensionality Reduction:** Dimensionality Reduction is a technique used to reduce the number of features in a dataset while retaining as much information as possible. The higher the dimensions are the more complex the computation becomes and the less efficient the analysis becomes which is generally known as Dimensionality Curse.
- **Singular Value Decomposition (SVD):** SVD is factorization of a matrix into three matrices which is used to get k latent semantic features by dimensionality reduction. It converts the matrices into square matrices and then performs Eigen Decomposition to get the latent features.
- **Latent Semantics:** Latent semantics is the resultant space we achieve when we map our object-feature matrix into latent semantic space by dimensionality reduction.
- **Distance or Similarity measures:** To compare the magnitude of similarity or dissimilarity between two or more data points, we use various distance or similarity measures. Some of them include euclidean, manhattan, Cosine similarity, dot product etc. The distance measure computes how much one data point is dissimilar to another whereas the similarity measure computes how much one data point is similar to another datapoint.

Task 2:

- **DBScan Algorithm:** DBScan algorithm is used for identifying clusters of data points in the dataset based on the dataset. It creates a neighborhood graph and includes a datapoint into the coreset if the number of neighbors in that graph is

greater than minimum count. It segregates data points into border point, noise point, core point.

- **Core Point:** A data point is considered as core point if it has a minimum number of neighbors within the threshold radius.
- **Border Point**: A data point is considered as border point if it is within the threshold radius but doesn't have a minimum number of numbers to become a core point itself.
- **Noise Point**: Data points that are neither core nor border point are noise points.

Task 3:

- **KNN Classifier or K- Nearest Neighbour Classifier:** KNN or K-Nearest Neighbour classification algorithm is where an object is classified based on the majority vote of k most similar, already classified objects.
- **Decision Tree Classifier:** Decision Tree Classifier works by recursively partitioning the dataset into subsets based on the values of features. The decision tree structure consists of nodes representing tests on features, branches representing the outcomes of the tests, and leaf nodes representing the final predicted class labels for classification. The tree is built in a top-down recursive manner, where the dataset is split based on the feature that provides the best information gain (entropy) or Gini impurity reduction or Fisher's discriminant ratio.
- **Personalized Pagerank Classifier or PPR Classifier:** Personalized PageRank classifier or PPR classifier assigns importance scores to nodes in a graph based on their influence relative to a set of seed nodes. These personalized scores are then used as features for training a classifier, enabling tasks such as classification, typically in the context of graph-structured data.

Task 4:

- **Locality Sensitive Hashing:** LSH is a method for hashing similar data points to the same "buckets" with high probability, allowing for efficient approximate similarity search in large datasets. It is commonly employed in scenarios where traditional exact search methods become computationally impractical due to the dimensionality of the data.
- **Hash Functions:** LSH employs hash functions that group similar items into the same hash buckets, with a higher likelihood of collision for similar items. The function used to partition data points is known as hash function and in this project we are using a window size of w to partition the data as we are using euclidean distances. These hash functions helps to remove false positives from the search result.
- **Layers:** A layer in LSH is a set of hash functions. In each layer, a different hash function is used to project the data points into hash buckets. The final result is obtained by considering the outcomes from all layers. Multiple layers allow for refining the search and reduce misses from the final result.

Task 5:

- **Relevance Feedback Systems:** Relevance feedback aims to improve the accuracy of search or recommendation results by iteratively incorporating user feedback.
- **Support Vector Machine (SVM):** Support Vector Machine (SVM) works by finding the hyperplane that best separates different classes in the feature space. SVM is particularly effective in high-dimensional spaces and is well-suited for scenarios where clear decisions of boundaries are needed. It works on the principle to maximize the margin between different classes while minimizing classification errors.
- **Support Vectors:** Support vectors are the data points closest to the decision boundary which plays a crucial role in defining the hyperplane.
- **Probabilistic Relevance Feedback System:** Probabilistic relevance feedback is used to enhance the accuracy of results by incorporating probabilistic models based on user feedback and iteratively updating these models based on user interactions.

Goal Description

Task 0a: The goal of task 0a is to implement a program which computes and prints the inherent dimensionality of the training dataset i.e. even numbered images in the Caltech 101 dataset.

Task 0b: The goal of task 0b is to implement a program which computes and prints the inherent dimensionality for each label in the training dataset i.e. even numbered images in the Caltech 101 dataset.

Task 1: The goal of task 1 is to familiarize with dimensionality reduction by implement a program which computes the k latent semantics by performing dimensionality reduction on the training data and then predicting the most likely labels for any test data with the latent semantics generated using distance/ similarity functions. After predicting the labels, the program should also display the per-label precision, recall, F1-score and overall accuracy of the prediction using the latent semantics.

Task 2: The goal of task 2 is to familiarize with clustering algorithms by implementing a program which uses the DBScan algorithm to compute the c most significant clusters based on the training data, even images. The clusters obtained from the above operations should be visualized in 2 ways, one of them is by representing them as differently colored point clouds in a two dimensional Multi Dimensional Space and other as groups of image thumbnails. Using these c clusters, the program has to predict the most likely labels for any test data. After predicting the labels, the program should also display the per-label precision, recall, F1-score and overall accuracy of the prediction happening using these clusters.

Task 3: The goal of task 3 is to familiarize with classification algorithms by implementing various classifier algorithms. In this task, we implement a program to classify train data using KNN classifier, decision tree classifier, Personalized Pagerank classifier and based on the resultant classification we predict the most likely labels for any of the training data based on any one of the user selected classifiers. After predicting the labels, the program also displays the per-label precision, recall, F1-score and overall accuracy of the prediction using these classifier algorithms.

Task 4a: The goal of task 4a is to familiarize with multi dimensionality indexing by implementing Locality Sensitive Hashing (LSH). Here, we implement a program that implements LSH on the train data based on the user selected number of hashes per layer and number of layers. This program should create an in-memory index structure containing the given set of vectors.

Task 4b: The goal of task 4b is to use the in-memory index to implement a similar image search algorithm and visualize the t most similar images retrieved. The program should also output the numbers of unique and overall number of images considered during the process.

Task 5: The goal of task 5 is to familiarize with classification algorithms and the effect of feedback on these classification algorithms. In this task, we write a program that implements a Support Vector Machine based relevance feedback system and a probabilistic relevance feedback system. Here, the program should use the output returned from task 4b and the feedback given by the user on the user selected feedback system.

Assumptions

We are assuming that the user already contains the caltech 101 dataset and the json files which consists of feature vectors and the label to image mapping data.

Task 1: We are assuming that the user provides the k value for which the program computes the k latent semantics.

Task 2: We are assuming the number of clusters given will be either 5 or 10.

Task 3: We are assuming that the user provides the m value for both mNN and PPR classifiers. In addition, we are using the RESNET50 Layer_3 feature model as the feature space for this task.

Task 4a: We are assuming that the user provides the value of number of Layers *l*, number of hashes per layer *h* and a RESNET50 Layers_3 features of even numbered images as input vectors.

Task 4b: We are assuming that the algorithm in this task will be using a previously computed in-memory index structure (stored in a pickle file). In addition, we assume that the user provides query_image from the set of odd numbered images and a value t.

Task 5: We assume the relevancy tags and data is such that the linear SVM is able to classify the data space.

Description of Proposed Solution/ Implementation:

Note: Instead of extracting the features again in phase 3, we are using the json file which contains the feature vectors for the train data from phase2. We are also using the json file which consists of label image mapping data i.e. it consists of a dictionary showing what are the images present in each label. Here, the image data present in both these files are computed only for train data to

avoid any discrepancies. We are doing this to improve the performance of the code by removing the feature computation time from the code.

Task 0a:

Design Decisions:

1. We used Multi Dimensional Scaling to compute inherent dimensionality. We preferred MDS over others, as we wanted to preserve the dissimilarities between various data points. One other reason to choose MDS is to get the minimum number of dimensions without a given dimension value, so that the dimensionality reduction can be inherent.
2. We used the breaking condition of MDS i.e. the point where the MDS algorithm should be stopped executing is when we have the minimum number of dimensions where the stress first becomes less than 0.05. We chose 0.05 as the value of stress for the breaking condition as it is described that if the stress value is less than or equal to 0.05 then the resultant representation is a good fit of the original dimensions in the lower dimensions. It is also described that if the stress is less than or equal to 0.025 then the resultant representation can be considered as an excellent fit for the original data but to break MDS at 0.025, we will need more iterations resulting in more computation making the algorithm expensive. So, we decided to go with a stress value less than or equal to 0.05 as the breaking condition. We also start from a random position for every dimension as a design decision.
3. To improve efficiency of the algorithm, we calculated the stress for dimensions in the intervals of 20. Whenever the stress becomes less than or equal to 0.05, we start searching for the minimum number of dimensions in that interval range of 20, where the uppermost dimension has stress less than 0.05 and the previously calculated lower dimension has stress greater than 0.05. We apply MDS for each dimension in between this interval and find the dimension where the stress first becomes less than 0.05 and output that dimension as the inherent dimensionality. This way the algorithm avoids calculating stress for the intervals where there is no chance of getting stress less than 0.05 and those dimensions will be pruned from the computation.
4. The number of iterations used to minimize stress at each dimension is taken as 300 which is known as the industry standard for Multi Dimensional Scaling. The transformation used to minimize the stress is Guttman transformation which takes into account the weight of distance in the lower dimension to the original dimension. We used this transformation, as it effectively reduces the stress by a larger amount even though we start from a random position for every dimension.

Solution

We selected layer_3 vector space to find the inherent dimensionality as they were giving better results for us in phase 2. We designed Multidimensional Scaling from scratch and used the stress function instructed by the professor in the lecture slides. The stress function in the slides normalizes the stress by dividing the stress with the sum of squares of original distances making sure that the stress can be comparable with the previous in terms of percentages for example 0.05 stress states

5% stress. When applying MDS, for any new dimension, we always start with random configuration in the specified dimension and calculate stress for this random configuration. After the stress is calculated, we check whether the stress is under 0.05, if it is then we stop the MDS algorithm and return the dimension as inherent dimensionality for the dataset otherwise we implement guttman transformation which takes weight into consideration when minimizing the stress by diving the lower dimensional configuration distances with the original distances of data. This weight ratio is then multiplied with the random configuration we used by taking mean resulting in new configuration of reduced distances in the same dimensions with minimized stress compared to before. This operation will be repeated for 300 iterations for each dimension until we find the inherent dimensionality. The metric used to calculate the distance between data points is L2-metric which is euclidean distance. Whenever the distance between two data points is 0 i.e. when we consider distance between the same image, we replace it with 0.00001 to avoid division by zero. The computation of inherent dimensionality for the entire train data is taking approximately 30 mins because of the large number of images. After computation, we are saving the inherent dimensionality value, resultant stress and reduced features into a json file for future reference.

Stress Computation:

```
def stress_function(original_distances, reduced_distances):
    stress = np.sqrt(((reduced_distances.ravel() - original_distances.ravel()) ** 2).sum() / ((original_distances.ravel() ** 2).sum()))
    return stress
```

Guttman Transformation:

```
# Guttman Transformation
reduced_distances[reduced_distances == 0] = 1e-5
ratio = original_distances / reduced_distances
B = -ratio
B[np.arange(len(B)), np.arange(len(B))] += ratio.sum(axis=1)
X = 1.0 / original_distances.shape[0] * np.dot(B, X)

reduced_distances = np.sqrt((X**2).sum(axis=1)).sum()
```

Breaking condition:

```
if current_stress < 0.05:
    print("breaking at iteration %d for dimensions %d with stress %s " % (iteration + 1, num_dimensions, current_stress))
    break
```

Random position initialization and max iterations values for specified dimension:

```
X = np.random.rand(num_images, num_dimensions)
# Set optimization parameters
max_iterations = 300
```

Output visualization: While displaying the output we are printing the stress value obtained at the end of 300 iterations in each dimension computed. Finally, the program also prints the overall inherent dimensionality for the train data along with the stress value obtained at that dimension.

Task 0b

Design Decisions:

1. We used Multidimensional Scaling to compute inherent dimensionality for each label. We preferred MDS over others, as we wanted to preserve the dissimilarities between various data points in each label. One other reason to choose MDS is to get the minimum number of dimensions, so that the dimensionality reduction can happen faster and be inherent.
2. We used the breaking condition of MDS i.e. the point where the MDS algorithm should stop executing is when we have the minimum number of dimensions where the stress first becomes less than 0.05. We chose 0.05 as the value of stress for the breaking condition as it is described that if the stress value is less than 0.05 or 5% then the resultant representation is a good fit of the original dimensions in the lower dimensions. It is also described that if the stress is less than or equal to 0.025 then the resultant representation can be considered as an excellent fit for the original data but to break MDS at 0.025, we will need more iterations resulting in more computation making the algorithm expensive and it may also not reduce dimensions to a desirable extent. So, we decided to go with a stress value less than 0.05 as the breaking condition.
3. To improve efficiency of the algorithm, we calculate the stress for dimensions in the intervals of 20. Whenever the stress becomes less than or equal to 0.05, we start searching for the minimum number of dimensions in that interval range of 20, where the uppermost dimension has stress less than or equal to 0.05 and the previously calculated lower dimension has stress greater than 0.05. We apply MDS for each dimension in between this interval and find the dimension where the stress first becomes less than 0.05 and output that dimension as the inherent dimensionality. This way the algorithm avoids calculating stress for the intervals where there is no need of calculating stress.
4. The number of iterations used to minimize stress at each dimension is taken as 300 which is known as the industry standard for Multi Dimensional Scaling. The transformation used to minimize the stress is Guttman transformation which takes into account the weight of distance in the lower dimension to the original dimension. We used this transformation, as it effectively reduces the stress by a larger amount even though we start from a random position for every dimension.

Solution

We selected layer_3 vector space to find the inherent dimensionality as they were giving better results for us in phase 2. To compute the inherent dimensionality for all the labels, we iterate over all the labels and apply MDS to all the images present in that label. We designed Multidimensional Scaling from scratch and used the stress function instructed by the professor in the lecture slides. The stress function in the slides normalizes the stress by dividing the stress with the sum of

squares of original distances making sure that stress is always in the range of 0-1. When applying MDS, for any new dimension, we always start with random configuration in the specified dimension and calculate stress for this random configuration. After the stress is calculated, we check whether the stress is under 0.05, if it is then we stop the MDS algorithm and return the dimension as inherent dimensionality for the dataset otherwise we implement guttman transformation which takes weight into consideration when minimizing the stress by diving the lower dimensional configuration distances with the original distances of data. This weight ratio is then multiplied with the random configuration we used by taking mean resulting in new configuration of reduced distances in the same dimensions with minimized stress compared to before. This operation will be repeated for 300 iterations for each dimension until we find the inherent dimensionality. The metric used to calculate the distance between data points is L2-metric which is euclidean distance. Whenever the distance between two data points is 0 i.e. when we consider distance between the same image, we replace it with 0.00001 to avoid division by zero. The computation of inherent dimensionality for all the labels is taking approximately 5 mins as the maximum number of images in each label is around 200. After computation, we are saving the inherent dimensionality value, resultant stress and reduced features of each label into a json file for future reference.

Stress Computation:

```
def stress_function(original_distances, reduced_distances):
    stress = np.sqrt(((reduced_distances.ravel() - original_distances.ravel()) ** 2).sum() / ((original_distances.ravel() ** 2).sum()))
    return stress
```

Guttman Transformation:

```
# Guttman Transformation
reduced_distances[reduced_distances == 0] = 1e-5
ratio = original_distances / reduced_distances
B = -ratio
B[np.arange(len(B)), np.arange(len(B))] += ratio.sum(axis=1)
X = 1.0 / original_distances.shape[0] * np.dot(B, X)

reduced_distances = np.sqrt((X**2).sum(axis=1)).sum()
```

Breaking condition:

```
if current_stress < 0.05:
    print("breaking at iteration %d for dimensions %d with stress %s " % (iteration + 1, num_dimensions, current_stress))
    break
```

Random position initialization and max iterations values for specified dimension:

```
X = np.random.rand(num_images, num_dimensions)
# Set optimization parameters
max_iterations = 300
```

Output visualization: While displaying the output we are printing all the labels and their inherent dimensionality value and the stress value obtained at that dimension.

Task 1

Design Decisions:

1. Dimensionality Technique : We tried with multiple dimensionality techniques and finally settled on SVD. As k is an input here, we can easily generate the k latent semantics using SVD
2. Comparison with label wise semantics : For each label, we calculate a mean label latent vector. We use this to compare our odd image features after it is projected to label wise latent space. The feature space that we are using here is layer 3 features extracted from RESNET.

Solution:

We implemented the SVD in a similar fashion as our previous phase implementation. We use the Eigen decomposition library to generate our eigenvalues and eigenvectors. Using this, we project our even images to latent space and save the latent space for transforming our odd images.

Task 2

Design Decisions:

1. We have used DBSCAN clustering to get the strongly connected components that, given the image features, clusters data with two parameters mincount and epsilon. The user inputs the number of clusters c and we can select the values of these parameters to output the specific number of clusters. For which mincout we have fixed in the range 1 to 6 as some labels have very few images which with increasing the mincount can lead to huge amounts of noise points. Some points can be common to two clusters (border points) which we decided to give to the cluster with a lesser number of points to promote balance. The epsilon value is found using binary search and the clusters c can be given with different combinations of mincount and epsilon but the combination of parameters with least number of noise points is selected.
2. To plot the clusters we have used MDS as we want to preserve distances between the images in two dimensional space which helps us understand the results of DBSCAN better by showing clusters as various colored points. We then also visualize the images thumbnails of the clusters to understand the clusters given by DBSCAN.
3. For a new odd image we first project it into the latent space which is SVD layer3 with 66 latent features then find the distance of this image from the core points of the specific clusters in the label. We will get the labels in which this image will not be a noise point, that is it may belong to, from which we can select one label with least distance to the image as

the most likely label. Note that we have not given multiple output labels as calculating accuracy, precision, recall, f1 scores came out to be difficult. Here, we have also handled gray odd images by increasing channels from one to three.

Assign Class (Core points):

```
def assign_class(density_dict, core_point, data_length, class_list, noise_point):
    # core_point.reverse()
    cluster_quant = {}
    for core in core_point:
        if class_list[core] == -1:
            class_list[core] = core
            cluster_quant[class_list[core]]=1
            density_propagation(density_dict, core, class_list, core_point, noise_point, cluster_quant)
    sorted_dict = dict(sorted(cluster_quant.items(), key=lambda item: item[1]))
    for k in sorted_dict.keys():
        visited = []
        # print(k)
        for epsilon_neib in density_dict[k]:
            density_propagation_non_core(density_dict, epsilon_neib, class_list, core_point, visited)
    return class_list
```

Binary Search of mincount and epsilon:

```
def binary_search_clusters(distances, mincount_low, mincount_high, epsilon_low, epsilon_high, target_clusters, max_itr=500):
  best_noise = distances.shape[0]
  best_clusters = 0
  while mincount_low <= mincount_high:
    itr = 0
    ep_l = epsilon_low
    ep_h = epsilon_high
    while itr<=max_itr:
        mid_epsilon = (epsilon_low + epsilon_high) / 2
        #print(f"Epsilon:{mid_epsilon}, mincount:{mincount_high}")
        class_list, core_point, noise_cnt = DBSCAN_main(distances, mds_data, epsilon = mid_epsilon, min_count = mincount_high)
        num_clusters = len(set(class_list)) - (1 if -1 in class_list else 0)
        itr+=1
        if num_clusters==target_clusters:
            print('best found at :',mincount_high, mid_epsilon)
        if num_clusters > best_clusters:
            best_clusters = num_clusters
            best_noise = noise_cnt
            best_eps = mid_epsilon
            best_mincount = mincount_high
        if num_clusters==0 or num_clusters > target_clusters or len(core_point) < target_clusters:
            epsilon_low = mid_epsilon
        else:
            epsilon_high = mid_epsilon - 1e-6

    # Reset epsilon range for the next iteration
    epsilon_low, epsilon_high = ep_l, ep_h
    mincount_high -= 1

  # # If no exact match is found, return the closest values
  # show_result(class_list, mds_data)
  print('noise: ',best_noise)
  return best_mincount, best_eps, class_list, core_point

```

Solution: To get the clusters in DBSCAN we choose mincount and epsilon as described above, we use mincount to check if the cluster has enough connected points to be considered a clustered and epsilon as a threshold distance value between images to consider them as connected or be the part of the same cluster. All the other points which may not satisfy these criteria will be considered as noise points. We start with a point (image) and sort the distances to check which points are below the threshold value and if the number of points are more than mincount we give it a cluster class, then we recur through all those connected points and find the connected points in the same way giving them the same class. If there are no more connected points in that cluster then we start with the rest of the points the same way until we have all the clusters, we call these sets of points as core points. This way all the points will be given a cluster class.

For many labels we were not able to get exactly 5 or 10 clusters as given by the professor, this is due to the fact that either the data points are not enough to create those many clusters or the data is very tightly bound that changing epsilon is helping with getting exactly that number of clusters. Although we have gotten 5 or 10 clusters for a lot of clusters and the data shows differences in the objects clustered in different clusters and that the clustering is done well enough to separate the data.

Visualization: We are visualizing a two dimensional plot and displaying the images in the clusters of the labels.

Task 3

Solution:

We have implemented following three classifiers as part of this task,

- **KNN Classifier:** Given even numbered images and a value k, our primary objective is to build a KNN classifier that effectively learns to classify images to their respective label. It operates by calculating the distance (such as Euclidean distance) between a test data point and all points in the training set, selecting the closest 'k' neighbors, and then assigning the most frequent label among these neighbors to the test point. Below are the details of the solution implemented.

  Class KNNClassifier

  Initialization: The **"__init__"** method initializes the classifier with a user-specified k, the number of neighbors to consider.

  Storing Training Data: The **"fit"** method stores the training data and labels preparing the classifier for prediction.

  Distance Calculation: The **"euclidean_distance"** method computes the Euclidean distance between two instances, a fundamental operation in KNN for determining the similarity between instances.

  Neighbor Retrieval: This function **"get_neighbors"** retrieves the nearest neighbors for a test instance. It uses a heap data structure for efficient neighbor selection, which is crucial given the potentially large size of the training set.

  Majority Voting & Single instance Prediction: This function **"predict_classification"** implements the core KNN logic, where the label of a test instance is determined based on the most common label among its nearest neighbors.

  Batch Prediction: The "**predict"** method handles batch prediction for test instances.

- **Decision Tree Classifier:** Given even numbered images and a value k, our primary objective is to build a Decision tree classifier that effectively learns to classify images to their respective label.We have experimented with multiple feature split criterias, maximum depth of the tree, minimum number of samples to perform the split, the type of features used for the split (full feature space or latent feature space), the number of features to be considered and changing the precision of the features. We found the best result to be using the gini impurity for the feature split and max depth of 40 with 25 as the minimum number of samples to perform the split. We used the layer 3 feature space

to form the decision tree.

Formula for gini impurity : 1- $\sum_{i=1}^{k} p_i^2$ (where k is the number of labels)

- **Personalized Pagerank Classifier:** Given even numbered images and a value k, our primary objective is to build a PPR based classifier that effectively learns to classify images to their respective label. To implement a PPR-based classifier, we begin by creating a similarity graph where each image in the dataset is compared with all others to determine similarity, and for each image, create edges in the graph with the n most similar images (similar to how we did in Phase 2). Next, calculate the personalized PageRank for each label by identifying the subset of nodes corresponding to that label and running the PageRank algorithm on the graph with this subset as the personalization vector. Afterward, we identify prototypes for each label by selecting the m nodes with the highest PageRank scores. Finally, build the classifier which, for a new instance, calculates its similarity with each prototype, estimates the similarities of the instance belonging to each class based on these similarities, and assigns the instance to the class with the mean of the similarities. Below are the details of the solution implemented.

  Class PPRClassifier

  Initialization: The **"__init__()"** method sets up the classifier with parameters like the number of nodes (n), the number of representatives per label (m), damping factor (beta), maximum iterations for convergence, and tolerance for convergence checking. It also initializes necessary attributes for the graph structure, labels, and label representatives.

  Training Data and Graph Construction: The **"fit()"** function takes training features and labels as input. It computes similarities among features and constructs a graph where nodes represent images, and edges represent similarities. This graph is used in the Personalized PageRank computation.

  Similarity Computation: This method **"compute_similarities()"** normalizes the features using StandardScaler and calculates pairwise similarities using cosine similarity.

  Graph Construction: In this method **"compute_graph()"**, we construct a graph with images as nodes. Each node connects to its n most similar nodes, with edge weights inversely proportional to n. This adjacency matrix is a key component in the PageRank algorithm.

  PageRank Calculation: This method **"personalized_pagerank()"** implements the Personalized PageRank algorithm. It iteratively updates the PageRank values based on the graph structure and a personalization vector, which represents the specific interest in certain nodes (images) for a given label.

Identifying Label Representatives: For each label, this method **"compute_label_representatives()"** computes a personalization vector that prioritizes images belonging to that label. Then, it applies the Personalized PageRank to find the top m representatives for each label.

Single Instance Prediction: This method **"predict_classification"** predicts the label of a single test instance by comparing it with the representatives of each label using cosine similarity and selecting the label with the highest average similarity.

Batch Prediction: This method **"predict()"** handles batch prediction for multiple test instances. It computes the label for each test instance as in predict_classification.

Design Decisions:

While implementing the classifiers, we made some design decisions described below:

KNN Classifier:

1. **Choice of Distance Metric – Euclidean Distance:**
    - Euclidean distance is a natural choice for measuring similarity in many applications due to its intuitive geometric interpretation. In image classification, it effectively quantifies the pixel-level difference between images.
    - **Implication**: This choice assumes that the feature space is a Euclidean space, where the straight-line distance between points is meaningful. It works well when features have similar types of scales but might not be optimal if the feature space is heterogeneous or highly dimensional.
2. **Use of a Heap Data Structure for Neighbor Selection**:
    - Using a heap optimizes the process of finding the k-nearest neighbors. It allows efficient insertion and extraction of distances, maintaining the k smallest distances encountered.
    - **Implication**: This design choice improves the efficiency of the neighbor search, particularly important for large datasets. It reduces the time complexity from O(nlogn) for sorting all distances to O(nlogk), where n is the number of training instances and k is the number of neighbors.
3. **Majority Voting for Classification**:
    - The classifier uses a simple majority voting scheme among the k-nearest neighbors to determine the label of a test instance. This approach is straightforward and effective for aggregating the predictions of neighbors.
    - **Implication**: Majority voting is robust to noise among the neighbors, as it relies on the most common label. However, it doesn't account for the relative distances of the neighbors or their potential differing reliabilities.

Decision Tree Classifier:

1. **Choice of Feature Splitting Criteria:**
    - After experimenting with both gini impurity and the entropy based on the industry standards, gini impurity gave the better results and also resulted in faster construction of the tree
2. **Choice of Decision Tree parameters:**
    - We experimented with multiple criterias and based on the best results, the following values were found:
        i. Maximum depth = 40
        ii. Minimum samples for split = 25
        iii. Precision of feature values = 0.01
        iv. Feature space used = Original features of resnet layer 3
        v. Number of features = All features
        vi. Threshold values for splitting = Unique values of feature

PPR Classifier:

1. **Design decisions for Personalized PageRank algorithm:**
    a. Given that each image is linked to only its 'n' most similar images, the adjacency matrix 'M' mostly contains zeros, especially for smaller values of 'n'. This sparsity provides us with an opportunity to use sparse matrix representations. For this reason, we converted our adjacency matrix to its corresponding sparse matrix representation and achieved more efficient computations and faster matrix operations.
    b. The personalization vector, denoted as S, is initialized such that all image entries belonging to the label ‘l’ are set to 1. The rest of the entries in this vector are zero. This vector is then normalized to ensure that the sum of its values is 1. We do this because Personalized PageRank is designed to provide rankings that are tailored to a particular node or set of nodes. The personalization vector serves as a “bias” or "restart" distribution: whenever the random surfer decides to jump to a random node (which happens with probability β), they choose according to the distribution defined by S. By setting entries of images with label 'l' to 1, and normalizing, we're biasing our rankings towards these images, thus ensuring the results are personalized with respect to label 'l'.
    c. We have started with equal probabilities for all nodes (i.e., setting the initial PageRank value for each node to 1/n) to ensure a uniform distribution. This means initially, each node is considered equally important. As the algorithm progresses, these initial equal probabilities will be modified based on the graph's structure and the personalization vector.
    d. The damping factor β (which is usually denoted as $1-\alpha$, where α is often set to around 0.85) represents the probability that the random surfer will jump to a random node in the graph. The value of 0.15 is a typical choice and was found by trial and error in the original PageRank research by Google's founders. It provides a good balance between considering the graph's actual structure and the random jumping behavior to ensure the convergence of the algorithm.
    e. We have used the Cosine Similarity metric to compute the Similarity Graph. We have tried with different distance metrics like Correlation distance, Euclidean distance and

Manhattan distance. We have found better results with Cosine Similarity. As in our case, the adjacency matrix representation of the graph being very sparse, cosine similarity works well with sparse vectors because it inherently accounts for the non-zero dimensions.

2. **Design decisions for PPR Classifier:**
    a. Deciding on n, the number of similar images to consider for each image in the graph, affects graph density. A denser graph might capture more nuances but could also increase computational complexity. We have chosen ‘n’ to be 200 for which we got better accuracy scores.
    b. For calculating the a combined similarity between the test instance and the label representatives, we have performed a pairwise cosine similarity for each of those representatives and computed a mean similarity score representing the similarity between the test instance and the label. By averaging, the effect of any single highly similar (or dissimilar) representatives is mitigated, which can lead to more stable and generalizable classification decisions. The mean is suitable when you want to capture the average similarity level of an instance to a class, providing a more balanced view.

Visualization:

We visualize the query image and display the predicted label. In addition, we also display the per-label precision, recall and f1-scores along with the overall accuracy score.

Task 4a

Solution:

In this section, we will discuss the implementation of the Locality-Sensitive Hashing (LSH) algorithm. LSH is a technique used for approximate nearest neighbor search in high-dimensional spaces. Our implementation is designed to efficiently index and retrieve similar data points from a large dataset.

LSH Class

We have implemented the LSH algorithm using a Python class called LSH. This class has the following key components:

Initialization:

- **L and h parameters**: The number of layers (L) and the number of hash functions per layer (h) are initialized when creating an instance of the LSH class. These parameters determine the trade-off between retrieval accuracy and efficiency.
- **Input vectors:** The class takes a set of input feature vectors as its input data.

Index Creation:

- **create_index():** This method initializes random hyperplanes for each layer, computes the bucket width W, and populates hash tables with data points. The choice of W plays a crucial role in determining the bucket assignment.
- **init_random_hyperplanes():** In this method, random hyperplanes are generated for each layer and hash function. These hyperplanes are used to project data points into buckets.
- **get_bucket_key():** This method computes the bucket key for a given vector and layer by projecting the vector onto the hyperplanes.
- **project_on_hyperplane():** Given an input vector and an LSH hyperplane, this method calculates the projection of the vector onto the hyperplane.
- **assign_vector():** Assigns a vector to a bucket based on its projected value and the bucket width W.

Query and Retrieval:

- **query():** Given a query vector, this method retrieves data points from the hash tables that share the same bucket keys as the query.
- **get_adjacent_buckets():** Calculates adjacent bucket keys for a given bucket key in a specific layer.
- **expanded_lsh_query():** Performs an expanded query by including adjacent buckets to increase the number of retrieved data points. This is useful when the initial query results are insufficient.

Usage: To use the LSH class, one can create an instance, call create_index() to build the index, and then perform queries to retrieve similar data points.

Design Decisions:

- **The L and h parameters:** These parameters are chosen based on the specific requirements of the application. Increasing L and h improves retrieval accuracy but may also increase computation time.
- **Bucket width W:** The choice of W is determined as a fraction of the minimum distance from the origin to any of the hyperplane vectors. This value affects the granularity of bucket assignments.
- **Random Hyperplanes:** We use random hyperplanes to ensure that data points are distributed across buckets in a way that minimizes collisions.
- **The expanded_lsh_query() method:** We provide an option to expand the search by including adjacent buckets. This is particularly useful when the initial query results do not meet the desired threshold.

Visualization: No visualization is required for this part as we are only creating the LSH tool, which is utilized in the next part. For this part, we are storing the index structure into a pickle file which can then be loaded for use during the query process in Task 4b.

Task 4b

Solution:

This section discusses the implementation of a similar image search using an LSH index structure. The goal is to efficiently retrieve images that are similar to a query image from a large dataset. The implementation leverages the LSH index built on image feature vectors to achieve this task.

Function similar_image_search_lsh()

The key component of the implementation is the similar_image_search_lsh() function, which performs the following steps:

1. **Feature Extraction**: It begins by extracting feature vectors for the query image using a pre-trained ResNet-50 model. The feature vectors are then used as the query.
2. **Candidate Selection**: The LSH index is queried using the query vector, and an initial set of candidate image indices is obtained. To enhance the search, the function performs an expanded LSH query by including adjacent buckets, further increasing the candidate pool.
3. **Candidate Filtering**: To ensure that there are enough candidates, the function checks if the number of candidates is less than the desired threshold t. If so, it performs a brute-force search by computing Euclidean distances to all data points in the dataset and selects the top t most similar images.
4. **Distance Calculation**: If the number of candidates is sufficient, the function calculates the Euclidean distances between the query vector and the candidate vectors. This step is crucial for ranking the candidates by similarity.
5. **Ranking and Output**: The candidates are ranked based on their distances to the query vector, and the top t most similar images are returned as results. The function also provides information about the total number of candidates considered and the total number of candidates after expanding the search.
    a. Total number of candidates during the query process is determined to be a union of query results across the L layers.
    b. Unique number of candidates is determined by performing a set operation on the union of candidates.

Design Decisions:

1. **Feature Vector Extraction:** We used ResNet50 to extract feature vectors from images. Specifically, we extracted feature vectors from the Layer3 of ResNet50. The reason for

extracting vectors at runtime is to handle the odd images, for which we do not have feature vectors.

2. **LSH Query:** For a given query image, we use the LSH index to find candidate images. The query vector is the feature vector extracted from the Layer3 of ResNet50. To ensure that there are t similar results returned, the function checks if the number of candidates is less than the desired threshold t. If so, we perform a brute-force search by computing Euclidean distances to all data points in the dataset and selecting the top t most similar images.
3. **Distance Calculation:** We calculate the Euclidean distances between the query vector and candidate vectors to find the most similar images.

Visualization: For the visualization, we are first visualizing the original(input) image, followed by the number of unique images considered, and the id's of the 't' most similar images. Then we display the 't' similar images.

Task 5

Design Decisions:

1. SVM based Relevance Feedback: We have decided to perform query expansion, where initially we ask LSH to give us t images and then t images every time the loop runs. Then we re rank the images based on given and predicted feedback. We have used one vs all Linear SVM technique as the number of classes are not binary thus for 4 relevant classes namely very relevant, relevant, irrelevant, and very irrelevant we use 4 classifiers. This system is for both even and odd images, and we use resnet50 layer 3 output for the image as features.
2. Probabilistic Relevance Feedback System: We have pre-processed all numerical descriptors to boolean ones, by picking a threshold value for each feature. This threshold value is picked as mean + standard deviation of that feature, if the value of the feature is greater than this threshold we give it a value of 1 and 0 otherwise. We then computed four kinds of probabilities of each feature.P(fi = 0|rel), P(fi = 1|rel), P(fi = 1|irrel), P(fi = 1|irrel).
    a. e number of objects marked as relevant by the user is 1 and the feature fk does not occur or is not sufficiently dominant in this single relevant
    b. object (i.e., rk = 0), the term becomes log(0) = −∞. To prevent this, p(fk|R) and p(fk|I) are often ap-proximated as

$$p(f_k|R) = \frac{r_k + 0.5}{|R| + 1} \quad \text{and} \quad p(f_k|I) = \frac{(d_k - r_k) + 0.5}{|D - R| + 1},$$

c. Since we have 4 categories we gave 0.75 contribution towards relevancy and irrelevancy respectively and 1 for very relevant and very irrelevant things.

Solution:

1. SVM based Relevance Feedback: In this task we used the output of previous task LSH output. Here the user will ask us to output 't' images given a query image. We then ask the user to give relevance feedback on the output images which will be given to Linear SVM for training. Here, we use one vs all technique, hence for four classes of relevance feedback we have used four SVMs. Then we expand our query search to get more than t images from the LSH, note here that we might encounter an image for which there are not enough images in the LSH buckets that we have also mentioned in Task4. After training the SVM we predict the class of the new images we got by expanding our search, from which we will sort the results based on the relevancy tags given to the both trained and test images, the ranked images will also be sorted by their distance from the query image but relevancy will be given priority. Thus our final results contain images re-ranked which were tagged relevant previously and test images. We do this on loop until the user is satisfied with the results or the search space is empty, but for each loop we ask the user to give feedback on the output images so that we can improve SVM training.

   Although the results were not very good, we were able to re-rank and identify relevant images based on the user feedback. This task was not able to yield very good results due to two main reasons, first as the training data was sparse, especially the very relevant images tags that the SVM was not able to train properly and training on subjectivity of the user on such sparse data is difficult. Second, that the LSH was not able to provide enough good relevant images from which the user can predict the relevant images, due to which we either get stuck at sub par images or the same results are shown again. Nevertheless, the results still showed promise as the SVM is showing relevant tags for few images and they are correct in that sense. Thus it is able to understand the subjectivity of the user and some relation between relevant images tagged by the user.

2. Probabilistic Relevance Feedback System: In this task we used the output of previous task LSH output. Here the user will ask us to output 't' images given a query image. We then ask the user to give relevance feedback on the output images which will be used to determine the weights of features with which each feature contributes towards this query. These weights are calculated using probabilities calculated according to the following formula:

$$log\left(\frac{p(f_k|R)(1-p(f_k|I))}{p(f_k|I)(1-p(f_k|R))}\right)$$

System Requirements/ Installation and execution instructions

Hardware Requirements

Processor: Quad-core processor or higher

Memory: 8 GB Ram or higher

Software Requirements:

Operating Systems: Windows 10 or later, Mac OS 10 or later

Libraries and Technologies: Python >= 3.7, NumPy >= 1.19.2, torch>=1.11.0, torchvision = 0.12.0, sciPy = 1.5.2, scikit-learn (distances), Pillow, matplotlib, PIP, pandas, Pickle, CV2

Execution Instructions

Meet all the above system requirements given above and install the above dependencies using pip**.**

The dataset will be downloaded using Torchvision package in the task 0a code in the previous directory under the data directory if it is not already downloaded.

The user should have json files - features.json, label_image_map.json and SVD_66_layer_3_latent_weights.json before executing the programs. These files should be present in the root directory where the dataset is downloaded and code is present.

The IDE we are using for this phase is **Google Colab** and this needs access and connection setup to google drive before executing the actual program. The access for this Collab has to be requested by the user and any one of the members from our team will approve this access within 24 hours. In every task the first cell indicates code to mount your google drive and once the user runs this cell, the google drive will be mounted. After doing this, the user has to create a shortcut to the root folder so that the Colab can know how to access these files which are present in the shared folder. Another way to access the code without google drive access is to unzip the phase-3 folder we submitted and download code of each task from the code folder and open the respective task in either Google Colab or Conda jupyter and run each one of them. For Jupyter please note that the libraries should be installed and the first cell to mount the drive needs to be removed/commented.

The code program file inside the code folder, cd into this folder.

Task0a: After opening task0a.ipynb you have to run all the cells present in that file. The last cell will display the output of the program.

Task0b: After opening task0b.ipynb you have to run all the cells present in that file. The last cell will display the output of the program.

Task1: After opening task1.ipynb you have to run all the cells present in that file. The last cell will ask for input k which represents the number of latent semantics and after the input is given the program will display the output of the program.

Task2: After opening task2.ipynb you have to run all the cells present in that file. The last cell will ask for input c which represents c most significant clusters and after the input is given the program will display the output of the program.

Task3: After opening task3.ipynb you have to run all the cells present in that file. The last cell will display the output of the program.

Task4a: After opening task4a.ipynb you have to run all the cells present in that file. The last cell will display the output of the program.

Task4b: After opening task4b.ipynb you have to run all the cells present in that file. The last cell will display the output of the program.

Task5: After opening task5.ipynb you have to run all the cells present in that file. The last cell will display the output of the program.

There may be some commented out code at the end of the code files, these were used for testing purposes so please ignore them.