COMP3204-CW2

Henry Card (hc3g21), Ayush Varshney (av3g21),
James Martin (jdm1g21), Simran Sandhu (ssss1g21)

January 10, 2024

1 General Structure

For this project, all functions are run within the main function of main.py. This is to make it easier by using the same code to import the data for each run, and by having global variables apply to all three runs.

2 Run 1

The first run is based off of the k-nearest neighbours algorithm to categorize images by calculating the distance between the vectorised input images. We achieved this through a variety of processes called within our 'imgToVector' function, starting with a process that involves cropping the images to a square around their centre and resizing them to a fixed resolution of 16x16 pixels, as per the specification. These vectors are then flattened from two dimensions to one using a concatenation function to combine the rows. This resulting vector is then standardised in a two step process. Initially, each mean is set to zero by subtracting the mean value from each value in the vector, this process mean-centres the vector. Following this, the vector is converted into a unit vector through unit length normalisation.

Once the image has been converted into vector format, the algorithm calculates the Euclidean distance between the input vector and the set of all training images across the different categories in vectors produced the same way. It then selects a specified K vectors that have the closest distance and determines which category the input vector is via majority voting from the selected vectors.

In terms of training the model, we used the 'train' function to convert all the images into vectors, following the previously described method, and then store them into separate text files. This data is then ready for classifications. We then used the 'test' function to perform image categorisation based on the K-nearest Neighbour algorithm previously implemented. Finally, we used the 'tune' function to automate the process of tuning the hyperparameter K to get the model to perform optimally. It does this by iterating through all possible values of K and evaluating them, returning the best performing value. We then used the 'testAccuracy' function to evaluate the accuracy of the classifier by testing against a validation set consisting of a subset of the training data. In the end we found that our best value for K was 25, which produced a score of 38.6% on our training data.

3 Run 2

In this run the aim was to develop a set of linear classifiers based on Bag of Visual Words. This was done by first creating a function to extract features from the images. It involves extracting

8x8 patches from each image using a stride of every 4 pixels. Each patch is then standardized by subtracting its mean and dividing by its standard deviation. This is required to normalise the data and improve the classifier performance. Finally the patches are flattened into vectors, turning each 8x8 patch into a 64x1 vector.

Next, we created a sampling function called ‘select’, which randomly selects a specified number of vectors from the vectors provided, in this case the vectors previously created. This random sampling is useful for reducing computation in the next step. We used K-means clustering to cluster the sampled patch vectors. The number of clusters then became the size of our visual vocabulary. This is an essential step for Bag of Visual Words as it quantises the image features into discrete words. After this, we worked on histogram creation, achieved using the ‘createArrays’ and ‘oneArray’ functions. After clustering, each patch in the image is assigned to the nearest cluster or visual. The histogram is then created representing the frequency of each visual word in an image, which is the feature vector used for classification.

In addition to this we created a function that implements TF-IDF. The ‘getInverseFrequencies’, this function calculates the mean frequency of each visual word across all categories. To normalise the mean frequencies into a range of 0 to 1, we used ‘py5.remap’. The function then subtracts the normalised value from 1 and raises the result to the power of 10. This step is crucial, as it inverts the frequency measure, so less frequent words (which are potentially more distinctive) get higher values. The adjusted frequencies is then used to update the histograms which now shows that less frequent bins are more informative. The process of generating these histograms is the what trains our model.

Finally, the ‘classify’ function was created to classify a given test image. First, the histogram of the test image would be computed then it would be compared with the histograms of different categories using cosine similarity, the category with the highest similarity score is chosen as the classification for the test image.

In terms of tuning our mode, the hyperparameters we could adjust consisted of patch size, stride length and the number of clusters in the K-means algorithm. We iterated over these variables and obtained the following values as the optimal hyperparameters for our model: PATCH_SIZE = 8, STRIDE_LENGTH = 4, SAMPLES = 200000, CLUSTERS = 2000. Here, the samples and clusters were significantly higher than our initial variables, while the stride length and patch size remained the same. We also tuned the power used within ‘getInverseFrequencies’ function, finding that 10 was the optimal value. After the hyperparameter tuning we managed to achieve an accuracy of 58.66% on our validation set obtained by splitting the training data.

4 Run 3

Training

Descriptor extraction

For this run, we decided upon using GIST descriptors as they offer a condensed representation of each image, reducing the dimensionality to enable higher computational speed and higher efficiency, especially for larger datasets. GIST features are robust, invariant to changes in viewpoints and illumination. A study by Sravani Yajamanam et al.[3] claimed that, when compared to deep learning techniques, GIST produced “equally strong results” and claimed that GIST may be “more robust” than deep learning algorithms, further supporting our choice of algorithm.

We followed the methodology outlined in Aude Oliva and Antonio Torralba's report [2] to learn and understand the computation process of GIST descriptors. We used this knowledge to implement their method in 'GIST.py'. The initial step involves creating a filter bank comprising of 32 Gabor filters, representing 8 orientations * 4 scales. Subsequently, the images are converted to grayscale, a Gaussian filter is applied, and they are resized to 256x256 pixels while maintaining the original aspect ratio. Following this, the filter bank is applied to each image. At this stage, each processed image is divided into a 4x4 grid and the mean of each grid cell is calculated. Each cell mean represents a feature which is concatenated with each other to form a descriptor for each image. Given the 32 filters and 16 cells for each image, the descriptors each contain 512 features.

In run3.py, GIST.py is used to extract the descriptors for the training images where they were stored with their corresponding labels in a CSV file. The image names were omitted since they were not needed to train the classifier.

Classification

GIST features work well when paired with a variety of algorithms, one of which being Support Vector Machines(SVM). SVMs allow for robust classifications with their optimal decision boundaries, generalising well to new data. This minimises the risk of overfitting, making them efficient at classifying new images accurately. The results of a study done by Peng Liu et al.[1] show that "in most cases SVMs are better than SAE", in the context of image classifications where SAE is a type of deep learning technique called a 'study auto-encoder'. These results further support our choice of algorithm. Overall, GIST features along with an SVM classifier provides an effective and reliable algorithm for image classification, making it an ideal choice for this run.

We trained the descriptors using an SVM with a radial basis function kernel. For this, the feature values are first normalised using 'StandardScaler' which is crucial for SVMs. The normalised data is then split into training and testing using stratified 10-fold cross-validation. Finally, the SVM classifier is trained on each training fold and the model's performance is evaluated on the corresponding test fold.

Tuning

Testing the accuracy of our model

To understand the effects of changing the model parameters, we needed to find a way to measure our model's performance. For this, we employed stratified 10-fold cross-validation to evaluate accuracy. This involves dividing the training images into 10 subsets, ensuring that each subset maintains group sizes proportional to the entire training set. By averaging the accuracy across 10 different splits, we obtain a reliable measure of the model's performance as this strategy is less sensitive to the impact of any single split. Stratification also minimises the likelihood of discrepancies between the classifier trained on the complete dataset and those trained on subsets.

Descriptor extraction

Recognising descriptor extraction as the most time-consuming section in run3, we implemented multiprocessing to distribute the descriptor extraction across several cores. Each folder in the training set was assigned to a different core, significantly reducing the time required to obtain results while experimenting with different parameters. Despite the efficiency gained from multiprocessing, we decided on a manual trial-and-error approach. Manually tuning the parameters allowed us to make informed decisions on the next parameters to try. Ultimately, we found that the optimal parameters include 11 orientations and 6 scales.

Classification

We created a function to iterate through various parameter combinations to tune the SVM. The specific parameters that were changed were the ‘kernel’, ‘C’ for regularisation strength and ‘gamma’ that influences the decision boundary shape. We explored three different kernels: radial basis function, polynomial and sigmoid. For the C values, we tested every value between 0.1 and 100 in increments of 0.1. For gamma, we experimented with ‘automatic’, ‘scale’ and a few common numerical values. The ‘find_best_svm_params’ function tested every combination to find that a radial basis function kernel, C value of 2.0 and an automatic gamma value were the best parameters.

5 Statement

Every team member has equally contributed to this project. All the runs were developed in a group coding style and any additional work was evenly split between the group.

References

[1] Peng Liu, Kim-Kwang Raymond Choo, Lizhe Wang, and Fang Huang. Svm or deep learning? a comparative study on remote sensing image classification. *Soft Computing*, 21:7053–7065, 2017.

[2] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, May 1 2001.

[3] Sravani Yajamanam, Vikash Raja Samuel Selvin, Fabio Di Troia, and Mark Stamp. Deep learning versus gist descriptors for image-based malware classification. In *Icissp*, pages 553–561, 2018.