Lauren Bright
Sophie Makaridi
Yanqi Liu

General Overview of Process

This project was performed in Google Colab using Python. The first step was to load the datasets, Cho and Cifar10 into a notebook from Google Drive. After the datasets were loaded into the notebook in array form, they were cleaned and normalized as necessary. In the Cifar dataset, there were no noisy data points to clean, and all data points were within the [0,255] range, so normalization was not needed. In the cho dataset, the first column was the gene_id and the second column was the ground truth values. These first two columns were removed so that only the values within each data point were left, our datasets now consisted of only rows of data points. The min/max method was then used to normalize the Cho sets so that certain attributes did not dwarf or overpower smaller attributes. Each row was transformed into new values within the range [0,1]. Now the data was ready to be used in the algorithms.

After both datasets were in a clean, normal form, they were divided into data and label datasets. The label datasets held the correct classifications for each data point in the data dataset. The KFold python library was then used to split the datasets into different training and testing sets. The parameter “n_splits” was set to 5, so KFold provided five different, unique combinations of training and testing sets per run. The “random_state” parameter of the KFold method was used to set the seed to yield a specific division of the data for each run. A different seed number was chosen for each of the five runs, so that the algorithms could be tested on five different arrangements of training/testing set combinations. Within each “fold” (an iteration of KFold), a model for the chosen algorithm was initiated and fit to the training and test sets of that fold. After the model was trained, using only the training set information, it was used to predict the values of the test set. Those predictions were then compared to the analogous test set values, and evaluated using accuracy, AUC, and precision. For each of the five iterations of KFold, the algorithm model was initiated, trained, and used to predict values from the test set. At the end, the evaluation values from each iteration were averaged to produce the values for that run. After all 5 runs, the average of all runs was taken to get the overall performance of the algorithm.

Random Forest

The random forest algorithm is considered an ensemble, tree-based algorithm because it combines more than one algorithm of the same kind for classifying given objects. The random forest classifier generates a set of classification trees, each of which is used to classify an object and “vote” for the class of the given object. The overall forest, the collection of decision trees, then chooses the classification for the object. This process is referred to as majority-vote, where each tree contributes one vote, and the class with the most votes wins as the classification for the object.

Every tree in the forest is built from samples of the training set. When constructing the trees, the splits on each node are determined by the input features. On their own, a single decision tree can often overfit and have high variance. But, by taking the average of a multitude of trees within the forest, some of the errors cancel out, so an overall reduced variance can be achieved. A reduced variance leads to a better overall model. In this implementation, the sklearn.ensemble.RandomForestClassifier, the algorithm combines the predictions of different trees by averaging their probabilistic prediction, not just letting each tree vote for only one class, but allowing different probabilistic weights to contribute to several classes. This approach is different from the majority-vote described above, but allows for more precise predictions from each tree.

For this project, the number of trees used was 100. This means that the training data was run on 100 different trees, and 100 sets of probabilistic predictions were yielded. Those predictions were averaged to find a final prediction classification for each object fed to the algorithm. Using more trees could provide perhaps more accurate overall predictions, but the running time was exceptionally long given the size of the CIFAR dataset.

K-nearest Neighbors

K-nearest neighbor is a simple algorithm that classifies new cases based on similarity measures. It assumes that “similar things exist in close proximity.” The general flow of the implementation of the k-nearest neighbor is as follow:

1) initialize K as the number of neighbors;
2) for each example in the training data set, calculate the distance between the test data and each row of the training data. In this step, Euclidean distance is the most popular distance metric to use;
3) sort the distances calculated in the previous step;
4) select the top K rows of the sorted array;
5) adopt “majority voting” from those K rows, which assigns the most common class label, to classify the data point.

One of the challenges in K-nearest neighbor classification is choosing the K value. Choosing the right K value is critical as it could yield higher accuracy and fewer errors. Different data sets have different K values that work best for them. A common strategy is to run the algorithm multiple times with various K values and see which one produces the best results. This was performed in this project.

Support Vector Machine

Goal of the support vector machine is to find a hyperplane in an N-dimensional space, so that it distinctly classifies the data points. Out of all possible hyperplanes that correctly classify data, SVM then looks for an optimal hyperplane with maximum margins. Margins in this case are distances to the points closest to the hyperplane. The algorithm uses a cost function to update gradients at every step and get a better classification.

When the data is not linearly separable, which in almost all cases it is not, it is useful to use Kernel functions. In my implementation, I used the Radial Basis Function (RBF). Using RBF, a support vector machine creates additional features to increase dimensions, so that points that were hard to classify linearly, become easily separable in higher dimensions. Some parameters to think about with my implementation of the algorithm were: kernel - which method of nonlinear separation to use, gamma - how significant new features/dimensions have on the decision boundary (used with RBF).

Evaluation (accuracy, AUC, recall)

SVM(CHO)

	Accuracy	AUC	Recall
1	0.7308	0.8469	0.7609
2	0.7013	0.8112	0.7036
3	0.7662	0.8567	0.7733
4	0.8442	0.8859	0.8114
5	0.8052	0.8694	0.7801
Average	0.7695	0.8417	0.7658
Standard Deviation	0.0510	0.0461	0.0348

SVM(CIFAR)

	Accuracy	AUC	Recall
1	0.5386	0.7031	0.5387
2	0.5423	0.7052	0.5421
3	0.5572	0.7088	0.5568
4	0.5441	0.7060	0.5445
5	0.5492	0.7071	0.5456
Average	0.5455	0.7082	0.5470
Standard Deviation	0.0510	0.04611	0.0349

KNN(CHO)

	Accuracy	AUC	Recall
1	0.7564	0.8326	0.7329
2	0.7403	0.8407	0.7481
3	0.7143	0.8213	0.7166
4	0.7013	0.7966	0.6691
5	0.6753	0.7726	0.6294

Average	0.7383	0.8282	0.7248
Standard deviation	0.0213	0.0122	0.0198

KNN(CIFAR)

	Accuracy	AUC	Recall
1	0.3364	0.394	0.3359
2	0.3381	0.3974	0.3375
3	0.3393	0.3913	0.3391
4	0.3319	0.3851	0.3340
5	0.3365	0.3875	0.3360
Average	0.3388	0.6327	0.3389
Standard deviation	0.0043	0.0016	0.0027

RANDOM FOREST(CHO)

	Accuracy	AUC	Recall
1	0.7564	0.8602	0.7822
2	0.6753	0.8014	0.6920
3	0.7403	0.8307	0.7278
4	0.7273	0.8004	0.6707
5	0.7532	0.8306	0.7239
Average	0.7305	0.8246	0.7193
Standard Deviation	0.0295	0.0222	0.0378

RANDOM FOREST(CIFAR)

	Accuracy	AUC	Recall
1	0.4582	0.6991	0.4585
2	0.4670	0.7037	0.4666

3	0.4671	0.7073	0.4677
4	0.4694	0.7051	0.4691
5	0.4589	0.6994	0.4589
Average	0.4641	0.7023	0.46412
Standard Deviation	0.0046	0.0025	0.0045

Comparing Results

Accuracy: The fraction of correct predictions a model produces. This metric is computed by simply dividing the number of correct predictions by the total number of predictions made.

AUC: The AUC, Area Under the Curve comes from an ROC (Receiver Operating Characteristic Curve). An ROC plots the true positive rate against the false positive rate, giving different classification thresholds at different points along the curve. The AUC is used to find an aggregate measure of performance among all classification thresholds on the curve.

Recall: A measure of how many actual positives a model correctly labels as positive. This metric is computed by dividing the number of correctly predicted positives by the number of total positive predictions.

Considering all three algorithms, the results for the Cho dataset were significantly higher than the results for the CIFAR dataset. There are several factors that could contribute to this difference, such as type, size, and quality of the data sets. Cho's data consists of numerical attributes and CIFAR's data consists of image data. Perhaps the image data is more difficult to train and form accurate predictions given the large number of pixels in each image. Cho only had a few hundred data points, whereas CIFAR had 60,000 data points. While we expected to see better performance from a significantly larger dataset, our results consistently showed Cho, the smaller dataset, to be better. One factor could be that the quality of the data from the Cho dataset was much higher than CIFAR, so no matter how many data points we had, there was more quality data to use from the smaller set.

Pros and Cons of Three Algorithms

Random Forest

Pros:

- Runs efficiently on large datasets
    - This we observed when comparing the runtimes of our three algorithms. Random Forest handled the larger CIFAR dataset much faster than KNN and SVM.
- The ensemble nature of the Random Forest algorithm reduces the sometimes high variance and overfitting of just a single decision tree
- Very flexible with the number of variables or attributes of a dataset

- Less affected by noise, or additional data points as that data point only impacts one tree, a small part of the entire forest voting for a classification.

Cons:

- Depending on the number of trees used in the algorithm, decision trees can tend to overfit on some datasets
- The number of trees used must be decided before running the algorithm. To get the best results, this can require trial and error of increasing the number of trees and observing how much the results improve.
- Creating an entire forest of trees requires more computational power and resources than some other algorithms
- The model can be complex and difficult to interpret due to the magnitude of decision trees used. Instead of looking at just one decision tree and looking at the different branches, there could be thousands of trees, making it difficult to interpret all of those trees.

K-nearest Neighbours

Pros:

- It is very simple to understand and easy to implement.
- No assumptions about the data because it is non-parametric.
- It has relatively high accuracy.
- It is versatile - useful for classification or regression.

Cons:

- It is computationally expensive at the prediction stage on large datasets.
- It is time-consuming when running on large datasets.
- It has a high memory requirement as it stores all or almost all of the training data.
- It is sensitive to outliers.
- It is sensitive to irrelevant attributes that would affect distance.

Support Vector Machine

Pros:

- Better overall results (Accuracy, Recall, AUC)
- Performs well on smaller datasets
- Effective in high dimensional space (many features), even when more features than samples
- Efficient with memory
- Outliers have less effect

Cons:

- Training takes a very long time with large datasets
- Does not perform as well with noisy data, when target classes overlap
- No direct probability estimates, needs kfold for that
- Need to carefully select kernel functions, performance varies greatly
- Prone to overfitting if way more features than samples

Findings (parameter sensitivity analysis)

K-nearest neighbors:
The only parameter for KNN is k, which represents the number of neighbors. During the parameter adjustment phase, I found that k=9 for CHO yields the best results in terms of accuracy, AUC, and recall. When it comes to CIFAR, the value of k seems less important in affecting the overall results. (Also, due to the extremely long runtime of CIFAR, I was not able to test for many rounds on the entire dataset. Instead, I performed parameter analysis based on a portion of the dataset.)

Support vector machine:
One of the most important parameters is Kernel functions: linear, poly, rbf.. I used rbf it gave the best results. Rbf increases dimensionality to better separate the data that is not linearly separable. Data and results were not too sensitive to the other parameters such as degree, tolerance, gamma so their default values were more effective than other values.
On large datasets, SVM takes a significantly long time, it was harder to test parameters, an easier way I found was to take a smaller part of the dataset and run tests on that, and after figuring out an optimal configuration, then run it on the whole dataset.

Random forest:
The main parameter modified in this implementation was n_estimators. This parameter controls the number of decision trees used in the classification. The final number was n_estimators = 100. Although the Cho dataset is smaller and could handle a larger number of trees, the CIFAR dataset's size made adding more trees to the equation increase the time drastically. Random Forest algorithms are known to handle large datasets well, but using a larger number of trees, like 1000, increased the time of each iteration significantly without major improvement in accuracy, AUC, or recall.