Combining Multiple Decision Trees with SLP

1st Ahmet Burak Yıldırım
dept. Computer Science
Özyeğin University
Istanbul, Turkey
burak.yildirim@ozu.edu.tr

2nd Arman Hasanzade
dept. Computer Science
Özyeğin University
Istanbul, Turkey
arman.hasanzade@ozu.edu.tr

***Abstract*—In this paper, we attempt to design a new classification algorithm by modifying some parts of the random forest algorithm which is an ensemble method. The first part that we modified is the training data selection of the individual trees running inside the random forest algorithm. We decided to train the individual trees with the training data belonging to a subset of the labels (classes). These subsets are selected by the genetic algorithm that we designed. Genetic algorithm aims to find the subset list of the labels that increases the accuracy of their corresponding individual trees. The second part that we modified is the decision algorithm of the random forest which is majority voting. We selected SLP and decision tree classifier as decision algorithms that take the prediction list of the individual trees and output the final prediction. The results of our experiment showed that increasing the accuracy of the individual trees by training with the data belonging to a subset of the labels caused overfitting. Because of that, the idea of using the genetic algorithm was not satisfactory. The results showed us, SLP decision algorithm is robust to the change of the individual tree models. Decision tree model as a decider algorithm was not a good idea because it could not decrease the priority of overfitted individual trees in the test phase. When the accuracies are compared with the other studies, we converged to the accuracies found. Due to the restricted time window, we could not implement additional features to our algorithm such as bootstrapping. Thus, our resultant accuracies are slightly lower than the compared studies.**

***Index Terms*—SLP, genetic algorithm, random forest, decision tree, ensemble method**

I. Introduction

In Machine Learning, there are many ensemble methods which utilizes a set of multiple learning algorithms to increase the accuracy of the predictions [1]. One of these methods is a classification algorithm called random forest. This algorithm operates by constructing a set of decision tree classifiers and training them with different subsets of the dataset to increase the variety of decision makers. After training each tree, the algorithm outputs the most occurring of labels (classes) predicted by the individual trees. This process is called majority voting [2]. As it is known that decision trees tend to overfit the training data, majority voting is used to decrease the effect of overfitting [3]. We aimed to come up with an algorithm in which we modify two aspects of the random forest classification algorithm. These are the training data selection for the individual decision trees and the voting algorithm used to reach a conclusion between the predictions of the individual trees.

In this paper, we decided to select different subsets of the labels to use in the training phase of the individual trees. The training data belonging to each selected subset is used to train one of the individual trees. In the label selection phase, we used genetic algorithm to increase the accuracy of individual trees. Genetic algorithm have been used in science and engineering as adaptive algorithm for solving practical problems and as computational models of natural evolutionary systems such as mutation, crossover and selection [4]. After selecting the labels, we trained the individual trees with respect to their selected label. After training the individual trees, we send the list of predicted labels as an input to three different decision algorithms for evaluating their performance. These algorithms are single layer perceptron (SLP), decision tree and majority voting. The output of the decision algorithm is set as the output of the ensemble method. After constructing the described algorithm, the performance of each modification idea is evaluated with three different datasets.

II. Methodology

In one of our first attempts to find more accurate individual trees, we have attempted to optimize the hyper parameters of the individual trees using the genetic algorithm. Our plan was to select random values for the parameters within a range that we would provide and train the trees using those parameters which increases the accuracy of the trees. We tried to apply some arithmetic operations to the hyper parameters of the most successful trees to create new ones with the help of genetic algorithm. However this approach was not successful because finding a range of the valid hyperparameters required unsatisfiable computational power and that made it impossible to run our algorithm in a feasible time. So we abandoned this idea and proceeded to find better ideas to try.

In this part, we will give the details of our algorithms in the following two main sections. In the label subset selection part, we will explain the genetic algorithm used while choosing the labels that increases the accuracies of the individual trees. In the decision algorithm section, we will give the in-depth implementation details of each algorithm used to decide output of the ensemble method.

A. Label Subsets Selection

As we mentioned before, we use genetic algorithm to decide which subset of the labels to use to train the individual trees.

In the initialization part, we create the first population of the subsets. Then, the evolution is done by selecting the multiple parent pairs of the next generation by giving higher selection probability to the individual trees having higher accuracy.

After selecting a parent pair, we use their label subsets in the crossover part to obtain the child subset of the parents. Before adding the child to the next generation, we mutate its labels with some probability. Finally, we add the child to the next generation and we do the same operations until the number of children is equal to the number of the current generation. In addition to the selection, we utilize elitism in this genetic algorithm. Elitism let's us pass the individual tree having the highest accuracy to the next generation directly as a child.

The children found become parents of the next generation and evolution continues until the aimed number of generations is reached. Ideally, the subsets of the labels in the last generation are the ones that make the accuracy of the individual trees higher than the previous generations. Thus, we used these labels during the training process.

1) Initialization: In this phase, we initialize the first population to apply the genetic algorithm steps on its members. Our algorithm has a hyper parameter named “label percentage” which denotes the number of labels to select for each subset list of the labels created to train the individual trees. For example, we have labels from 0 to 10 in the dataset. If the label percentage is 0.5, some of the possible subsets are: [3, 5, 8, 1, 2], [5, 9, 3, 6, 0] and [1, 2, 6, 9, 3] etc.

Another hyper parameter is “N” which is the number of individual trees to train in our algorithm (similar to random forest). Label subsets of the first population are generated with respect to the number of labels in a subset (Eq 1) for N trees.

$$\#\text{of labels in a subset} = \#\text{of labels in total} \cdot \text{label percentage} \tag{1}$$

2) Selection: In the selection process, firstly we train the individual trees inside the current population and calculate the accuracy of the trained models by using the validation data. Then we subtract the minimum accuracy from the accuracy of each individual tree and assign the result as the selection probability value for that tree. We divide each probability value to their sum to normalize the probabilities. Then, we select 2 parents (individual tree) from the population with the probabilities found and send them to the crossover process.

3) Crossover: In the crossover process, the parents which were selected by the selection step are used to create a new label subset (child). To create the child subset, we merge both parent subsets to ignore the duplicate labels and select random subsets having the same length with the subset of parents. The result is set as the subset of the labels to be used for the child individual tree.

4) Mutation: Labels which are not present in the label subset of the current child are found and stored. We also have a hyper parameter called “mutation rate” (it is a float value between 0 and 1). For each label in the subset of labels are mutated by chance with regards to the mutation rate. The label which is to be mutated is removed from the current subset and in its place, a random label is selected from the list of absent labels which was created at the start.

B. Decision Algorithm

After the label subset selection process, the individual trees are trained by their subset of the labels. Prediction labels of the individual trees are given as inputs to each of the following decision algorithms separately. The performance of these decision algorithms will be compared and discussed in the results and discussion section.

1) SLP: We construct a SLP model with the help of Pytorch library. The prediction list of the individual trees are one hot encoded and given as inputs to the SLP model. The input size of the SLP is found by the Eq 2 and the output size is set to the number of labels in total. The weights found for a one-hot-encoded prediction label unit, represents the importance of that unit when the connected output unit of the weight is the predicted label. So that, the explainability of our model is not affected by the SLP model.

$$\text{SLP input size} = \#\text{of labels in a subset} \cdot \#\text{of individual trees} \tag{2}$$

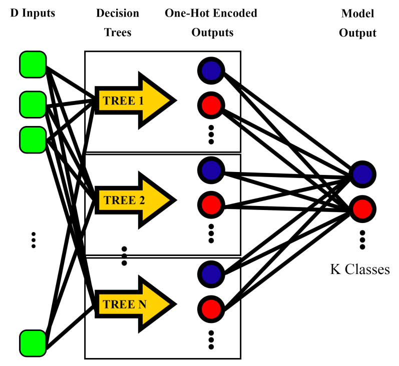

Fig. 1: SLP decider design

2) Majority Voting: As it is mentioned in the introduction part, this decision algorithm outputs the most occurring of labels (classes) predicted by the individual trees. It is the default decision algorithm of the random forest classification algorithm. The difference between our model and random forest is that we train each individual tree with data belonging to the subset of the labels. This decision algorithm is used to compare the training data selection process of our model and random forest classification algorithm.

3) Decision Tree: Similar to the SLP model, we connect the predicted label of the individual trees as input to a decision tree (named as prediction tree) without one-hot encoding them. The output of the prediction tree is set as the output of the ensemble model. In this case, we aimed to observe if the decision tree makes the model overfit to the training data by giving boundaries to each individual tree.

III. Experimental Setup

To conclude our experiments, we have used three different datasets which are MNIST, Fashion MNIST and Cifar-10. The

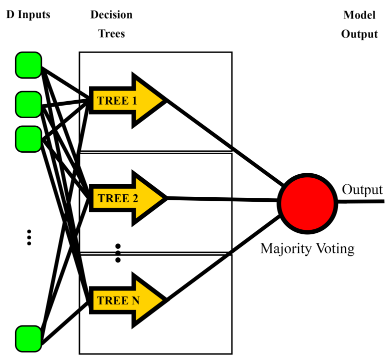

Fig. 2: Majority voting decider design

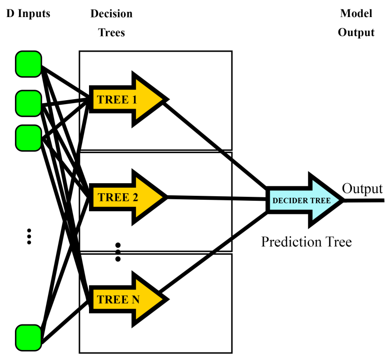

Fig. 3: Decision tree decider design

MNIST dataset consists of 60,000 28x28 pixel handwritten digit images. The Fashion MNIST has the same image properties as the MNIST dataset. However, instead of handwritten digit images, it contains images of different cloth types. The Cifar-10 dataset consists of 60,000 32x32 coloured images of different objects and animals. All three of these datasets have 10 different labels.

To observe the effect of our modifications, we focused on 3 different hyper-parameters which are generation number, N (number of individual trees) and label percentage.

We named our modified model as Custom Genetic Forest Classifier which contains the implementation of three different decision algorithms and the genetic algorithm that finds better individual trees. The individual trees are constructed by using scikit-learn's decision tree classifier with the default hyper-parameters of the library.

A. Generation Number

Ideally, we expect an increase in the average accuracy of the individual trees (named as population mean accuracy) as the new generations are created. In this experiment, we aimed to see the change in the accuracy of the population mean during the evolution. At the same time, we plot the accuracies of the models using 3 different decision algorithms that we previously mentioned. This is done to see whether the accuracy increase in the individual trees also increases the accuracies of our modified models or not.

In the setup, we set the value of the generation number to 10 in order to print the performances of each generation until 10th generation.

The hyper parameters of this setup are set as the values in Table 1.

Hyper Parameter	Value
N	100
Mutation Rate	0.2
Label Percentage	0.5

TABLE I: Generation number setup configuration

B. Number of Individual Trees (N)

We wanted to observe the effect of N on our models. In addition to the three previously mentioned models, we used the Random Forest Classifier of the scikit-learn library. The aim of this comparison is to find out how the majority voting decision algorithm (used in original random forest algorithm) performs without a restriction of a subset of the labels used to train the individual trees. To make the algorithm of the scikit-learn classifier similar to ours, we set the resample size of the algorithm to 0.5. The reason is that, because the label percentage of this setup is 0.5, our algorithm can use approximately 0.5 of the training data for each individual tree. We did not compare the scikit-learn's random forest classifier with default parameters because it makes optimizations that we can not implement in a restricted time. The hyper parameters of the setup for our model are set as the values in Table 2.

Hyper Parameter	Value
Class Percentage	0.5
Mutation Rate	0.2
Generation Number	5

TABLE II: Number of individual trees setup configuration

With the given configuration, modified models are trained with 3 different datasets and the results are plotted. Note that in Cifar-10 we could not find 5 generations due to computational power restriction. It is set to 1 to reduce the requirement.

C. Label Percentage

The aim of this setup is to see the effect of change in the size of the labels subsets used to train individual trees. Test sizes are decided as 0.2, 0.4, 0.6, 0.8 and 1. The hyper parameters of this setup are set as the values in Table 3. With the given

Hyper Parameter	Value
N	100
Mutation Rate	0.2
Generation Number	5

TABLE III: Label percentage setup configuration

configuration, modified models are trained with 3 different datasets and the results are plotted.

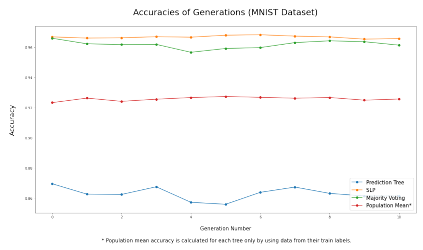

Fig. 4

IV. Results and Discussion

A. Generation Number

As it can be seen from Figure 4, using genetic algorithm on MNIST dataset has no major effect on the population mean (mean accuracy of individual trees). The reason can be the fact that MNIST is a dataset which can be easily classified (in generation 0 it is close to best) compared to the other ones. Thus, changing the subset of the labels by using genetic algorithm does not affect the accuracies as expected. The mutation rate can be decreased to strictly focus on the best trees. However it may result in overfitting. Even when the mean accuracies of the individual trees are not changing that much due to the change in the training data (generations) of the individual trees. The change in the individual tree models affects the accuracy of decision algorithms. The most affected decision algorithm is the prediction tree algorithm. Majority voting has been affected significantly in the interval of 3rd and the 7th generation. The most stable algorithm is the SLP which was not affected by the change of individual trees. It is adapted to the change easily.

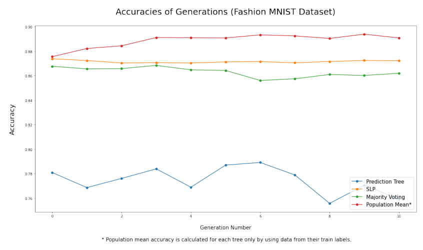

Fig. 5

In the Fashion MNIST dataset, genetic algorithm did a great job by increasing the mean accuracy of the individual trees. This can also be observed in Figure 5. The increase of the population mean can be caused by the separability of the classes but the Fashion MNIST dataset is harder than the MNIST dataset. Genetic algorithm finds the most separable subset of the labels so that the mean accuracy increases. However, it did not increase the accuracy of all 3 models. It seems like the variation of the individual trees is decreased and the random forest ensemble method becomes less useful to predict better. It is a type of overfitting. The stability of the accuracies over the generation change is similar with the MNIST results. Even when the accuracies decrease as the overfitting occurs, SLP is the most robust one to these type of cases.

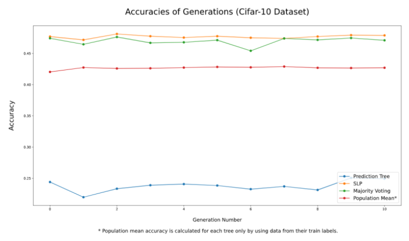

Fig. 6

In the Cifar-10 dataset, the genetic algorithm has not a significant effect as it can be seen in Figure 6. The reason can be that the subset labels of the Cifar-10 dataset are not separable as MNIST and Fashion MNIST. Thus the mean accuracy of the population is not affected by the subset selection of the genetic algorithm. The accuracy of the decider algorithms are not fluctuating as much as the other datasets. The reason is that overfitting cannot occur due to the complexity of the dataset. Individual trees can not increase their accuracy significantly.

When we consider the overall results, increasing the accuracy of the individual trees was not a good idea for the random forest model because it causes overfitting and the decider predictions get worse. Also, using genetic algorithm requires extreme computing power. As a result, the genetic algorithm result is not satisfactory.

B. Number of Individual Trees (N)

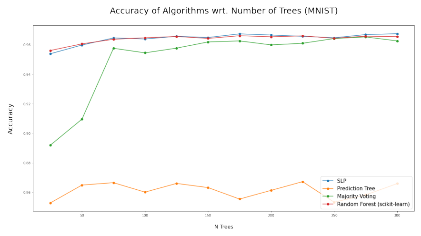

Fig. 7

Model accuracies over the change of the number of trees (N) can be seen from Figure 7 for the MNIST dataset. As scikit-learn uses the entire dataset for training its individual trees, using less number of trees did not affect its performance compared to the majority voting model which uses a subset of the labels to train its individual trees. Genetic algorithm may also cause overfitting more easily for the models having less number of trees because they use the label subsets with the best accuracy more than other labels. SLP, majority voting and scikit-learn's random forest algorithm converges to the same accuracy as the number of individual trees increase. The worst performance is caused by the prediction tree. As it is stated in the introduction part, decision trees tend to overfit. Using a prediction tree as a decider algorithm during the training phase, can again cause to overfit the training data. We can easily see that its performance is worse than the other methods.

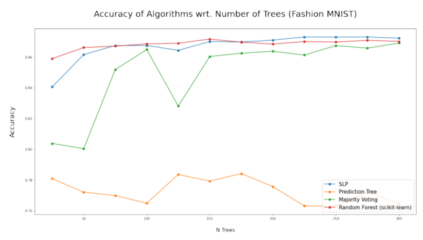

Fig. 8

In the Fashion MNIST dataset (Figure 8), the results and the comments are similar to the MNIST part.

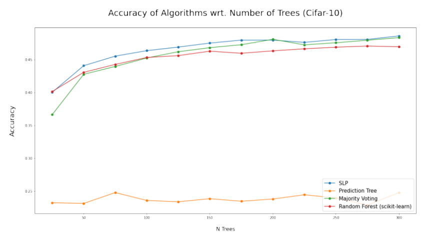

Fig. 9

In the Cifar-10 dataset (Figure 9), as it is stated in the experimental setup part, we did not use the genetic algorithm. So it did not cause overfitting for the lower N values. This can be also observed from the majority voting algorithm.

For overall comparison, slp is robust in the case of overfitting even if we use genetic algorithm or not. As it is known, for less number of trees, that scikit-learn does not use genetic algorithm it produces better performance compared to the overfitted ones. The accuracy always converges to a point as the number of trees increases. When we compare the results with other papers, random forest accuracy on the MNIST dataset converges to 97.0% [5], while our algorithm converges to 96.5%. As we stated in the methodology part, there are additional features of random forest such as bootstrapping that we could not implement in a restricted time window. But the accuracies are still close to the ones in the paper. As the result of another research [6] shows that Fashion MNIST and Cifar-10 converge to 88.4% and 52.0% accuracy respectively. Our results are 87.2% and 48.6% respectively. When the additional improvements in the paper are considered, our results are closer to the paper's result.

C. Label Percentage

The accuracy results of the 3 datasets can be seen from the Figure 10, 11 and 12, using 0.2 percentage of the labels (classes) decreased the overall accuracy (Example label subset: [2, 3]). The best accuracy is obtained by using the percentage values between 0.4 and 0.6. Using entire labels for training did not give better results but it should not be forgotten that we do not resample the data for the training process of the individual trees. So, the algorithm is not similar to the original random forest classifier when the class percentage is set to 1.0.

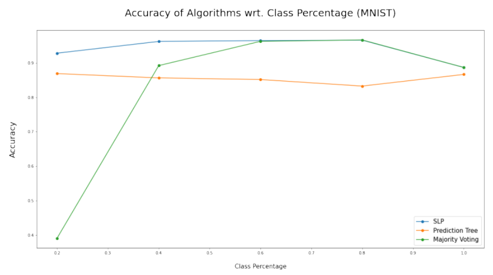

Fig. 10

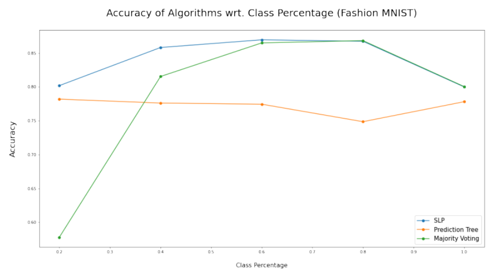

Fig. 11

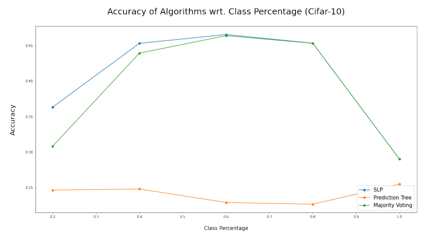

Fig. 12

V. Conclusion

We aimed to make different modifications on random forest classification algorithm. Firstly we tried to use genetic algorithm to predict better hyper parameters for the individual trees but it required computational power which was not feasible. Then we decided to use the data belonging to different subsets of the labels to train each individual tree. The subsets that increase the accuracies of the individual trees are selected with the help of the genetic algorithm. When we compared the results, the genetic algorithm overfit the data and did not give the expected result. On the other hand we used SLP, majority voting and prediction tree as a decider algorithm of the ensemble model. When the results are compared with different setups, SLP was the most robust decider to the change of the individual tree models. In general, the SLP algorithm was making more accurate predictions compared to the majority voting and prediction tree decider. The prediction tree decider algorithm overfit the data so its predictions were not good enough compared to the others. Accuracy of majority voting and SLP deciders converged as the number of individual trees increased.

REFERENCES

[1] Maclin, R., & Opitz, D. (1997). An empirical evaluation of bagging and boosting. In Proceedings of the Fourteenth National Conference on Artificial Intelligence, pp. 546– 551 Providence, RI Retrieved from https://arxiv.org/pdf/1106.0257.pdf

[2] Tin Kam Ho, ”Random decision forests,” Proceedings of 3rd International Conference on Document Analysis and Recognition, Montreal, Quebec, Canada, 1995, pp. 278-282 vol.1, doi: 10.1109/ICDAR.1995.598994. Retrieved from https://ieeexplore.ieee.org/document/598994

[3] Hastie, Trevor; Tibshirani, Robert; Friedman, Jerome, 2008, pp. 587–588. The Elements of Statistical Learning (2nd ed.). Springer. ISBN 0-387-95284-5.

[4] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA, USA, 1998. Retrieved from https://mitpress.mit.edu/books/introduction-genetic-algorithms

[5] B. Xu, Y. Ye and L. Nie, ”An improved random forest classifier for image classification,” 2012 IEEE International Conference on Information and Automation, Shenyang, 2012, pp. 795-800, doi: 10.1109/ICInfA.2012.6246927, RI Retrieved from https://ieeexplore.ieee.org/document/6246927

[6] Khoi Hoang, ”Image Classification with Fashion-MNIST and CIFAR-10”, California State University, Sacramento. RI Retrieved from http://athena.ecs.csus.edu/ hoangkh/Image%20Classification%20with%20Fashion-MNIST%20and%20CIFAR-10%20-%20Final%20Report.pdf

Source code of our project

January 20, 2021

```
[ ]: from mnist import MNIST
     import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn.tree import DecisionTreeClassifier
     from sklearn.model_selection import train_test_split
     from sklearn import metrics
     from sklearn.preprocessing import OneHotEncoder
     from threading import Thread
     from operator import itemgetter
     from sklearn.ensemble import RandomForestClassifier
     import torch
     import torchvision
     import torchvision.transforms as transforms
     import torchvision.datasets as datasets
     import torch.nn as nn
     import torch.nn.functional as F
     import torch.optim as optim
     import torch.optim.lr_scheduler as lr_scheduler
```

```
[ ]: class Perceptron(nn.Module):
         def __init__(self, d, K):
             super(Perceptron, self).__init__()
             self.model = nn.Linear(d, K)

         def forward(self,x):
             batch_size = x.shape[0]
             x = x.view(batch_size, -1)
             x = self.model(x.float())
             return x
```

```
[ ]: class CustomGeneticForestClassifier:
         def __init__(self, N, generation_number, class_percentage):
             self.N = N
             self.generation_number = generation_number
             self.class_percentage = class_percentage
             self.trained_trees = []
             self.slp = None
```

```
        self.prediction_tree = None
        self.one_hot_encoder = None
        self.SEED = 0
        self.EPOCHS = 20
        self.LR = 0.01              # learning rate
        self.MOMENTUM = 0.9         # momentum for the optimizer
        self.WEIGHT_DECAY = 1e-5    #
        self.GAMMA = 0.1            # learning rate schedular
        self.BATCH_SIZE = 32        # number of images to load per iteration

        self.train_tree_batch = 20
        self.mutation_rate = 0.2
        self.population_list = None

    def train_net(self):
        self.slp.train()
        epoch_loss = 0.0
        for xt, rt in self.train_loader:
            xt, rt = xt.to(self.device), rt.to(self.device)
            self.optimizer.zero_grad()
            yt = self.slp(xt)
            loss = self.loss_fn(yt, rt)
            loss.backward()
            self.optimizer.step()
            epoch_loss += loss.item()
        return epoch_loss

    def train_tree(self, filter_label, state_counter, tree_list):

        y_train_subtree = self.y_train[self.y_train.label.isin(filter_label)]
        X_train_subtree = self.X_train.loc[y_train_subtree.index.values.
↪tolist()]

        dtc = DecisionTreeClassifier(random_state=state_counter)
        dtc = dtc.fit(X_train_subtree,y_train_subtree)
        y_valid_filtered= self.y_valid[self.y_valid.label.isin(filter_label)]
        X_valid_filtered = self.X_valid.loc[y_valid_filtered.index]
        y_pred = dtc.predict(X_valid_filtered)
        tree_accuracy = metrics.accuracy_score(y_valid_filtered, y_pred)
        print(state_counter+1, end=" ")
        tree_list.append({"tree": dtc, "accuracy": tree_accuracy,␣
↪"filter_label": filter_label})

    def fit_trees(self, filter_labels, tree_list):
        print("Started to train {} trees.".format(len(filter_labels)))
        print("Trained Trees: ", end="")
        state_counter = 0
        train_threads = []
        for filter_label in filter_labels:
            train_threads.append(Thread(target=self.train_tree,␣
↪args=[filter_label, state_counter, tree_list]))
            state_counter += 1
        for thread_index in range(0, len(train_threads), self.train_tree_batch␣
↪):
            current_train_threads = train_threads[thread_index:
↪thread_index+self.train_tree_batch ]
            for train_thread in current_train_threads:
                train_thread.start()
            for train_thread in current_train_threads:
                train_thread.join()
        print("\n{} Trees are trained.".format(len(filter_labels)))

    def train_slp(self, one_hot_encoded_predictions):
        print("Started to train SLP.")
        self.d = one_hot_encoded_predictions.shape[1]      # number of input␣
↪features

        # manual seed to reproduce same resultsnet
        torch.manual_seed(self.SEED)

        self.slp = Perceptron(self.d,self.K)
        cuda = torch.cuda.is_available()
        self.device = torch.device("cuda:0" if cuda else "cpu")
        self.slp.to(self.device)
        self.loss_fn = nn.CrossEntropyLoss()
        self.optimizer = optim.SGD(self.slp.parameters(), lr=self.LR,␣
↪momentum=self.MOMENTUM, weight_decay=self.WEIGHT_DECAY)
        self.scheduler = lr_scheduler.StepLR(self.optimizer, step_size=10,␣
↪gamma=self.GAMMA)

        train_target = torch.tensor(self.y_train.values.flatten().astype(np.
↪int32)).long()

        train = torch.tensor(one_hot_encoded_predictions)

        train_tensor = torch.utils.data.TensorDataset(train, train_target)
        self.train_loader = torch.utils.data.DataLoader(dataset = train_tensor,␣
↪batch_size = self.BATCH_SIZE, shuffle = True, num_workers=8)

        # train the network
        for epoch in range(1,self.EPOCHS+1):
            self.train_net()
        print("SLP is trained.")
    def one_hot_encode(self, total_predictions):
        self.one_hot_encoder = OneHotEncoder(handle_unknown='ignore')
        self.one_hot_encoder.fit(total_predictions)

        one_hot_encoded_predictions = self.one_hot_encoder.
↪transform(total_predictions).toarray()
        return one_hot_encoded_predictions


    def train_prediction_tree(self, one_hot_encoded_predictions):
        self.prediction_tree = DecisionTreeClassifier(random_state=200)
        self.prediction_tree = self.prediction_tree.
↪fit(one_hot_encoded_predictions, self.y_train)


    def fit(self, X_train, y_train):
        self.X_train, self.X_valid, self.y_train, self.y_valid =␣
↪train_test_split(X_train, y_train, test_size=0.2, random_state=0)  #␣
↪Train-test split pairs
        self.label_count = len(y_train.label.unique())
        self.sample_count = y_train.shape[0]
        self.K = self.label_count                # number of output features

        self.population_list = self.genetic_find_parameters()
        last_population = self.population_list[-1]

        self.trained_trees = [member['tree'] for member in last_population]

        total_predictions = self.forest_trees_predict(self.X_train)
        one_hot_encoded_predictions = self.one_hot_encode(total_predictions)
        self.train_slp(one_hot_encoded_predictions)
        self.train_prediction_tree(total_predictions)

    def model_analysis(self, X_train, y_train, X_test, y_test):
        self.X_train, self.X_valid, self.y_train, self.y_valid =␣
↪train_test_split(X_train, y_train, test_size=0.2, random_state=0)  #␣
↪Train-test split pairs
        self.label_count = len(y_train.label.unique())
        self.sample_count = y_train.shape[0]
        self.K = self.label_count                # number of output features

        self.population_list = self.genetic_find_parameters()

        majority_accuracies = []
        slp_accuracies = []
        prediction_tree_accuracies = []
        population_mean_accuracies = []

        for population in self.population_list:

            population_mean_accuracy = np.mean(np.asarray([member['accuracy']␣
↪for member in population]))
            population_mean_accuracies.append(population_mean_accuracy)

            self.trained_trees = [member['tree'] for member in population]
            total_predictions = self.forest_trees_predict(self.X_train)
            one_hot_encoded_predictions = self.one_hot_encode(total_predictions)
            self.train_slp(one_hot_encoded_predictions)
            self.train_prediction_tree(total_predictions)

            majority_voting_pred = self.majority_voting_predict(X_test)
            slp_pred = self.slp_predict(X_test)
            prediction_tree_predict = self.prediction_tree_predict(X_test)

            majority_accuracy = metrics.accuracy_score(y_test,␣
↪majority_voting_pred)
            slp_accuracy = metrics.accuracy_score(y_test, slp_pred)
            prediction_tree_accuracy = metrics.accuracy_score(y_test,␣
↪prediction_tree_predict)

            majority_accuracies.append(majority_accuracy)
            slp_accuracies.append(slp_accuracy)
            prediction_tree_accuracies.append(prediction_tree_accuracy)
        generation_numbers = np.arange(len(self.population_list))
        return generation_numbers, prediction_tree_accuracies, slp_accuracies,␣
↪majority_accuracies, population_mean_accuracies


    def majority_voting_predict(self, X_test):
        total_predictions = self.forest_trees_predict(X_test)
        # Majority Voting
        predicted_values = []
        for row in total_predictions:
            majority_vote = np.bincount(row).argmax()
            predicted_values.append(majority_vote)
        y_pred_class = np.asarray(predicted_values)
        return y_pred_class

    def slp_predict(self, X_test):
        total_predictions = self.forest_trees_predict(X_test)
        # SLP
        one_hot_encoded_predictions = self.one_hot_encoder.
↪transform(total_predictions).toarray()
```

```
        test = torch.tensor(one_hot_encoded_predictions)
        y_pred = self.slp(test.to(self.device))
        y_pred = y_pred.cpu().detach().numpy()
        y_pred_class = np.asarray([np.argmax(pred) for pred in y_pred])
        return y_pred_class

    def prediction_tree_predict(self, X_test):
        total_predictions = self.forest_trees_predict(X_test)
        # Prediction Tree
        y_pred_class = self.prediction_tree.predict(total_predictions)
        return y_pred_class

    def forest_trees_predict(self, X_test):
        total_predictions = self.trained_trees[0].predict(X_test)
        for i in range(1, self.N):
            total_predictions = np.vstack([total_predictions, self.
↪trained_trees[i].predict(X_test)])
        total_predictions = np.transpose(total_predictions)
        return total_predictions


    # Genetic algorithm
    def generate_parent_samples(self):
        generation = []
        for i in range(self.N):
            generation.append(np.random.choice(range(self.label_count),␣
↪round(self.label_count*self.class_percentage), replace=False))
        return generation


    def genetic_find_parameters(self):
        print("Genetic algorithm is started.")
        generation = self.generate_parent_samples()
        population_list = []
        for i in range(self.generation_number+1):
            print("Generation:",i)
            trained_tree_results = []
            self.fit_trees(generation, trained_tree_results)
            population_list.append(trained_tree_results)
            generation = self.evolve(trained_tree_results)
        print("\nGenetic algorithm is ended.")
        return population_list


    def evolve(self, trained_tree_results):
        trained_tree_results_sorted = sorted(trained_tree_results,␣
↪key=itemgetter("accuracy"), reverse=True)

        next_generation = []

        # Elitism
        next_generation.append(trained_tree_results_sorted[0]["filter_label"])
        for i in range(1, len(trained_tree_results)):
            parent_1 = self.tournament(trained_tree_results)
            parent_2 = self.tournament(trained_tree_results)
            child = self.crossover(parent_1, parent_2)
            child = self.mutate(child)
            next_generation.append(child)
        return next_generation


    def crossover(self, parent1, parent2):
        parents_merged = np.unique(np.append(parent1, parent2))
        child = np.random.choice(parents_merged, len(parent1), replace=False)
        return np.sort(child)


    def mutate(self, child):
        mutated_child = []
        if len(child) == self.label_count:
            return child
        non_existing_labels = []
        for label in range(self.label_count):
            if label not in child:
                non_existing_labels.append(label)
        for gen in child:
            if np.random.random() < self.mutation_rate:
                if len(non_existing_labels) !=0:
                    selected_label_index = np.random.
↪randint(len(non_existing_labels))
                    mutated_child.append(non_existing_labels.
↪pop(selected_label_index))
            else:
                mutated_child.append(gen)
        return np.sort(mutated_child)

    def tournament(self, generation):
        accuracies = np.asarray([tree["accuracy"] for tree in generation])
        accuracies -= np.min(accuracies)
        probabilities = np.asarray(accuracies)/sum(accuracies)
        selected = np.random.choice(generation, 1,␣
↪p=probabilities)[0]["filter_label"]
```

```
        return selected
```

[ ]:
```
np.random.seed(60) # reproducability

def mnist_dataset_read(path):
    mndata = MNIST(path)

    # read training images and corresponding labels
    tr_images, tr_labels = mndata.load_training()
    # read test images and corresponding labels
    tt_images, tt_labels = mndata.load_testing()

    # convert lists into numpy format and apply normalization
    tr_images = np.array(tr_images) / 255. # shape (60000, 784)
    tr_labels = np.array(tr_labels)         # shape (60000,)
    tt_images = np.array(tt_images) / 255. # shape (10000, 784)
    tt_labels = np.array(tt_labels)         # shape (10000,)

    columns_images = ['p{}'.format(i+1) for i in range(784)]
    X_train = pd.DataFrame(data=tr_images, columns=columns_images)
    y_train = pd.DataFrame(data=tr_labels, columns=['label'])
    X_test = pd.DataFrame(data=tt_images, columns=columns_images)
    y_test = pd.DataFrame(data=tt_labels, columns=['label'])
    return X_train, X_test, y_train, y_test
```

[ ]:
```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

def load_data(btch):
    labels = btch[b'labels']
    imgs = btch[b'data'].reshape((-1, 32, 32, 3))

    res = []
    for ii in range(imgs.shape[0]):
        img = imgs[ii].copy()
        img = np.fliplr(np.rot90(np.transpose(img.flatten().reshape(3,32,32)),␣
 ↪k=-1))
        res.append(img)
    imgs = np.stack(res)
    return labels, imgs

def load_data_cifar():
    batch1 = unpickle("Datasets/cifar-10-batches-py/data_batch_1")
    batch2 = unpickle("Datasets/cifar-10-batches-py/data_batch_2")
    batch3 = unpickle("Datasets/cifar-10-batches-py/data_batch_3")
    batch4 = unpickle("Datasets/cifar-10-batches-py/data_batch_4")
    batch5 = unpickle("Datasets/cifar-10-batches-py/data_batch_5")
    test_batch = unpickle("Datasets/cifar-10-batches-py/test_batch")

    pixel_num = 32*32*3
    x_train_l = []
    y_train_l = []
    for ibatch in [batch1, batch2, batch3, batch4, batch5]:
        labels, imgs = load_data(ibatch)
        x_train_l.append(imgs)
        y_train_l.extend(labels)
    x_train = np.vstack(x_train_l)
    y_train = np.vstack(y_train_l)

    x_test_l = []
    y_test_l = []
    labels, imgs = load_data(test_batch)
    x_test_l.append(imgs)
    y_test_l.extend(labels)
    x_test = np.vstack(x_test_l)
    y_test = np.vstack(y_test_l)

    del batch1, batch2, batch3, batch4, batch5, test_batch

    x_train, x_test = x_train.reshape(-1, pixel_num), x_test.reshape(-1,␣
 ↪pixel_num)

    columns_images = ['p{}'.format(i+1) for i in range(pixel_num)]
    X_train = pd.DataFrame(data=x_train, columns=columns_images)
    y_train = pd.DataFrame(data=y_train, columns=['label'])
    X_test = pd.DataFrame(data=x_test, columns=columns_images)
    y_test = pd.DataFrame(data=y_test, columns=['label'])

    return X_train, y_train, X_test, y_test
```

```
[ ]: def plot_genetic_analysis(generation_numbers, prediction_tree_accuracies,␣
 ↪slp_accuracies, majority_accuracies, population_mean_accuracies,␣
 ↪dataset_name):
    plt.figure(figsize=(20, 10))
    plt.plot(generation_numbers, prediction_tree_accuracies, marker="o", label␣
 ↪= "Prediction Tree")
    plt.plot(generation_numbers, slp_accuracies, marker="o", label = "SLP")
    plt.plot(generation_numbers, majority_accuracies, marker="o", label =␣
 ↪"Majority Voting")
    plt.plot(generation_numbers, population_mean_accuracies, marker="o", label␣
 ↪= "Population Mean*")
```

```
    plt.title("Accuracies of Generations ({} Dataset)\n".format(dataset_name),␣
↪fontsize=25)
    plt.xlabel("\nGeneration Number\n\n * Population mean accuracy is␣
↪calculated for each tree only by using data from their train labels.",␣
↪fontsize=15)
    plt.ylabel("Accuracy\n", fontsize=20)
    plt.legend(loc='lower right', prop={'size': 15})
    plt.show()
```

```
[ ]: m_X_train, m_X_test, m_y_train, m_y_test = mnist_dataset_read('Datasets/MNIST')
     fm_X_train, fm_X_test, fm_y_train, fm_y_test = mnist_dataset_read('Datasets/
      ↪Fashion_MNIST')
     c_X_train, c_y_train, c_X_test, c_y_test = load_data_cifar()
```

```
[ ]: cgfc = CustomGeneticForestClassifier(N=200, generation_number=10,␣
      ↪class_percentage = 0.5)
```

```
[ ]: m_generation_numbers, m_prediction_tree_accuracies, m_slp_accuracies,␣
      ↪m_majority_accuracies, m_population_mean_accuracies = cgfc.
      ↪model_analysis(m_X_train, m_y_train, m_X_test, m_y_test)
```

```
[ ]: plot_genetic_analysis(m_generation_numbers, m_prediction_tree_accuracies,␣
      ↪m_slp_accuracies, m_majority_accuracies, m_population_mean_accuracies,␣
      ↪dataset_name = "MNIST")
```

```
[ ]: for i in range(len(m_generation_numbers)):
         print("Generation ", m_generation_numbers[i], ":", end=" ")
         print("\tSLP:{}\tMajority Voting:{}\tPrediction Tree:{}\tPopulation Mean␣
      ↪Accuracy:{}".format(m_slp_accuracies[i], m_majority_accuracies[i],␣
      ↪m_prediction_tree_accuracies[i], m_population_mean_accuracies[i]))
```

```
[ ]: cgfc = CustomGeneticForestClassifier(N=200, generation_number=10,␣
      ↪class_percentage = 0.5)
```

```
[ ]: fm_generation_numbers, fm_prediction_tree_accuracies, fm_slp_accuracies,␣
      ↪fm_majority_accuracies, fm_population_mean_accuracies = cgfc.
      ↪model_analysis(fm_X_train, fm_y_train, fm_X_test, fm_y_test)
```

```
[ ]: plot_genetic_analysis(fm_generation_numbers, fm_prediction_tree_accuracies,␣
      ↪fm_slp_accuracies, fm_majority_accuracies, fm_population_mean_accuracies,␣
      ↪dataset_name = "Fashion MNIST")
```

```
[ ]: for i in range(len(fm_generation_numbers)):
         print("Generation ", fm_generation_numbers[i], ":", end=" ")
         print("\tSLP:{}\tMajority Voting:{}\tPrediction Tree:{}\tPopulation Mean␣
      ↪Accuracy:{}".format(fm_slp_accuracies[i], fm_majority_accuracies[i],␣
      ↪fm_prediction_tree_accuracies[i], fm_population_mean_accuracies[i]))
```

[ ]:
```
cgfc = CustomGeneticForestClassifier(N=200, generation_number=10,␣
↪class_percentage = 0.5)
```

[ ]:
```
c_generation_numbers, c_prediction_tree_accuracies, c_slp_accuracies,␣
↪c_majority_accuracies, c_population_mean_accuracies = cgfc.
↪model_analysis(c_X_train, c_y_train, c_X_test, c_y_test)
```

[ ]:
```
plot_genetic_analysis(c_generation_numbers, c_prediction_tree_accuracies,␣
↪c_slp_accuracies, c_majority_accuracies, c_population_mean_accuracies,␣
↪dataset_name = "Cifar-10")
```

[ ]:
```
for i in range(len(c_generation_numbers)):
    print("Generation ", c_generation_numbers[i], ":", end=" ")
    print("\tSLP:{}\tMajority Voting:{}\tPrediction Tree:{}\tPopulation Mean␣
↪Accuracy:{}".format(c_slp_accuracies[i], c_majority_accuracies[i],␣
↪c_prediction_tree_accuracies[i], c_population_mean_accuracies[i]))
```

[ ]:
```
def plot_accuracies(x_values, accuracies, title, x_label):
    plt.figure(figsize=(20, 10))
    for algorithm, accuracies in accuracies.items():
        plt.plot(x_values, accuracies, marker="o", label = algorithm)
    plt.title(title, fontsize=25)
    plt.xlabel("\n{}".format(x_label), fontsize=15)
    plt.ylabel("Accuracy\n", fontsize=20)
    plt.legend(loc='lower right', prop={'size': 15})
    plt.show()
```

[ ]:
```
def get_accuracies(X_train, X_test, y_train, y_test, tree_number,␣
↪generation_number, class_percentage):
    cgfc = CustomGeneticForestClassifier(N=tree_number, generation_number =␣
↪generation_number, class_percentage = class_percentage)
    cgfc.fit(X_train, y_train)
    slp_pred=cgfc.slp_predict(X_test)
    prediction_tree_pred=cgfc.prediction_tree_predict(X_test)
    majority_voting_pred=cgfc.majority_voting_predict(X_test)

    slp_accuracy = metrics.accuracy_score(y_test, slp_pred)
    prediction_tree_accuracy = metrics.accuracy_score(y_test,␣
↪prediction_tree_pred)
    majority_voting_accuracy = metrics.accuracy_score(y_test,␣
↪majority_voting_pred)

    return slp_accuracy, majority_voting_accuracy, prediction_tree_accuracy
```

[ ]:
```
def number_of_trees_analysis(N_list, X_train, X_test, y_train, y_test):
    accuracies_of_N_trees_y = {"SLP": [], "Prediction Tree": [], "Majority␣
↪Voting": [], "Random Forest (scikit-learn)":[]}
```

```
    for N in N_list:
        slp_accuracy, majority_voting_accuracy, prediction_tree_accuracy =␣
↪get_accuracies(X_train, X_test, y_train, y_test, tree_number = N,␣
↪generation_number = 0, class_percentage = 0.5)
        rf = RandomForestClassifier(n_estimators = N, max_samples = 0.5)
        X_train_rf, X_valid_rf, y_train_rf, y_valid_rf =␣
↪train_test_split(X_train, y_train, test_size=0.2, random_state=0)
        rf.fit(X_train_rf, y_train_rf.values.ravel())
        rf_pred = rf.predict(X_test)
        accuracies_of_N_trees_y["Random Forest (scikit-learn)"].append(metrics.
↪accuracy_score(y_test, rf_pred))
        accuracies_of_N_trees_y["SLP"].append(slp_accuracy)
        accuracies_of_N_trees_y["Prediction Tree"].
↪append(prediction_tree_accuracy)
        accuracies_of_N_trees_y["Majority Voting"].
↪append(majority_voting_accuracy)
    return accuracies_of_N_trees_y
```

[ ]:
```
def class_percentage_analysis(class_percentage_list, X_train, X_test, y_train,␣
↪y_test):
    accuracies_of_class_percentages_y = {"SLP": [], "Prediction Tree": [],␣
↪"Majority Voting": []}
    for class_percentage in class_percentage_list:
        slp_accuracy, majority_voting_accuracy, prediction_tree_accuracy =␣
↪get_accuracies(X_train, X_test, y_train, y_test, tree_number = 100,␣
↪generation_number = 5, class_percentage = class_percentage)
        accuracies_of_class_percentages_y["SLP"].append(slp_accuracy)
        accuracies_of_class_percentages_y["Prediction Tree"].
↪append(prediction_tree_accuracy)
        accuracies_of_class_percentages_y["Majority Voting"].
↪append(majority_voting_accuracy)
    return accuracies_of_class_percentages_y
```

[ ]:
```
N_list = [i*25 for i in range(1,13)]
```

[ ]:
```
m_accuracies_of_N_trees_y = number_of_trees_analysis(N_list, m_X_train,␣
↪m_X_test, m_y_train, m_y_test)
```

[ ]:
```
plot_title = "Accuracy of Algorithms wrt. Number of Trees (MNIST)\n"
plot_accuracies(N_list, m_accuracies_of_N_trees_y, plot_title, "N Trees")
print(m_accuracies_of_N_trees_y)
```

[ ]:
```
fm_accuracies_of_N_trees_y = number_of_trees_analysis(N_list, fm_X_train,␣
↪fm_X_test, fm_y_train, fm_y_test)
```

```
[ ]: plot_title = "Accuracy of Algorithms wrt. Number of Trees (Fashion MNIST)\n"
     plot_accuracies(N_list, fm_accuracies_of_N_trees_y, plot_title, "N Trees")
     print(fm_accuracies_of_N_trees_y)
```

```
[ ]: c_accuracies_of_N_trees_y = number_of_trees_analysis(N_list, c_X_train,␣
      ↪c_X_test, c_y_train, c_y_test)
```

```
[ ]: plot_title = "Accuracy of Algorithms wrt. Number of Trees (Cifar-10)\n"
     plot_accuracies(N_list, c_accuracies_of_N_trees_y, plot_title, "N Trees")
     print(c_accuracies_of_N_trees_y)
```

```
[ ]: class_percentage_list = [i*0.2 for i in range(1,6)]
```

```
[ ]: m_accuracies_of_class_percentages_y =␣
      ↪class_percentage_analysis(class_percentage_list, m_X_train, m_X_test,␣
      ↪m_y_train, m_y_test)
```

```
[ ]: plot_title = "Accuracy of Algorithms wrt. Class Percentage (MNIST)\n"
     plot_accuracies(class_percentage_list, m_accuracies_of_class_percentages_y,␣
      ↪plot_title, "Class Percentage")
     print(m_accuracies_of_class_percentages_y)
```

```
[ ]: fm_accuracies_of_class_percentages_y =␣
      ↪class_percentage_analysis(class_percentage_list, fm_X_train, fm_X_test,␣
      ↪fm_y_train, fm_y_test)
```

```
[ ]: plot_title = "Accuracy of Algorithms wrt. Class Percentage (Fashion MNIST)\n"
     plot_accuracies(class_percentage_list, fm_accuracies_of_class_percentages_y,␣
      ↪plot_title, "Class Percentage")
     print(fm_accuracies_of_class_percentages_y)
```

```
[ ]: c_accuracies_of_class_percentages_y =␣
      ↪class_percentage_analysis(class_percentage_list, c_X_train, c_X_test,␣
      ↪c_y_train, c_y_test)
```

```
[ ]: plot_title = "Accuracy of Algorithms wrt. Class Percentage (Cifar-10)\n"
     plot_accuracies(class_percentage_list, c_accuracies_of_class_percentages_y,␣
      ↪plot_title, "Class Percentage")
     print(c_accuracies_of_class_percentages_y)
```