MACHINE LEARNING - 2021

Stud: Manuel-Alexandru Dragomir, group 246, ICA

Prof: Prof. Gabriela Czibula

Second software project

Github: https://github.com/manudragomir/random-forest-from-scratch

(4) experimental results and their analysis (doc)*;*

▪ *for evaluating the performance of the ML model*

- *a cross-validation will be used*
- *a statistical analysis of the results will be provided (e.g. confidence intervals)*
- *evaluation measures o for classification: accuracy, precision, recall, sensitivity, f-measure, AUC (Area under the ROC curve) o for regression: MAE (Mean of Absolute Errors), RMSE (Root Mean Squared Error), NRMSE (Normalized Root Mean Squared Error)*

Experiments:

RF (Random Forest) =

{*own*: own implementation of random forest, *skl*: sklearn implementation, *no:* no random forest – only decision trees}

(same for DT = Decision Trees)

Acc = Accuracy, Prec = Precision, Rec = Recall, Sens = Sensitivity

Mentions and Observations:

- first 3 experiments are extracted from previous laboratories;
- all experiments were done using criterion='gini'
- all experiments can be found under folder *experiments* from git folder, here we only centralised the most relevant results
- for cross validation we also added results about training with confidence intervals below results on test set
- we have done experiments using our random forest and sklearn decision classifier and we saw that the results are pretty similar with sklearn random forest (time and performances)
- regarding decision trees, the performances are similar, but our implementation is slightly slower as it can be seen;

	RF	DT	Acc	Prec	Rec	F1	Time	Other mentions
1.	skl	skl	0.9285	0.9397	0.9075	0.9191	74s	RF (n_estimators=1000, criterion='gini', max_depth=15, min_samples_split=2, min_samples_leaf=1)
2.	skl	skl	0.9212	0.9236	0.9212	-	-	same as above but with cross validation cv=5
3.	no	skl	0.7903	0.7914	0.7903	-	740s	DT (criterion='gini', max_depth=5, min_samples_split=2, min_samples_leaf=3) cv = 10
4.	own	skl	0.77	-	-	-	-	RF (n_estimators=10, max_depth=5, n_jobs=4)
5.	own	skl	0.9257	0.9329	0.9111	0.9219	204s	RF (n_estimators=500, max_depth=15, n_jobs=4) rf_10.01.2022_18_22.24
6.	own	skl	0.7477	0.7893	0.6974	0.7405	106s	RF (n_estimators=500, max_depth=5, n_jobs=4)
7.	own	skl	0.8851	0.8800	0.8717	0.8758	13s	RF (n_estimators=10, max_depth=15, n_jobs=4) rf_10.01.2022_18_16.21
7.1	own	skl	0.9091 0.9109±0.006	0.9091 0.9149±0.005	0.8932 0.8944±0.007	0.9011 0.9045±0.006	25s	RF (n_estimators=50, max_depth=15, n_jobs=4, cv=4) rf_11..01.2022_00.42.9
8.	own	own	0.8022	0.7946	0.7824	0.7884	97s	RF (n_estimators=3, max_depth = 15, n_jobs = 4, cv = 3) rf_11.01.2022_14:33.43
9.	own	own	0.7264 0.7276±0.002	0.7888 0.7748±0.004	0.6741 0.6822±0.008	0.7270	145s	RF (n_estimators=5, max_depth=5, n_jobs=4, cv=3) rf_11.01.2022_14.22.13
10.	own	own	0.8748	0.8762	0.8602	0.8681	415s	RF (n_estimators=10, max_depth=10, n_jobs=4, cv=0)
11	no	own	0.6636	0.6508	0.6399	0.6453	41s	DT (max_depth=5, max_thresholds=10) dt_10.01.2022_18.45.38
12.	no	own	0.7931	0.7835	0.7823	0.7833	110s	DT (max_depth=15, max_thresholds=10) dt_10.01.2022_18.48.05
13.	no	own	0.7990	0.7843	0.7858	0.7851	212s	DT (max_depth=15, max_thresholds=15) dt_10.01.2022_20.34.26

(5) the electronic version of the source code (including the data sets used for demonstration and the executable file) + a demonstration report (user manual)

Electronic version of the source code

The dataset and the code is uploaded here:
https://github.com/manudragomir/random-forest-from-scratch

The project can be round either with *python run.py* or launching the executable called *runnable.exe*. Both runnable.exe and run.py are placed in the root of the git directory.

User manual:

In this section, we are going to explain the main joints of our application. The app can be split in some major modules: **preprocessing, random forest module, decision tree module, utils, run.**

1. Preprocessing

The preprocessing part consists in preparing the data for the training. Mainly, the data can be found in two separate files *codon_usage.csv* which is the data we are going to train our models (aprox. 10000 samples) and *test.csv* which was mainly used to overfit our models for a sanity check (only 10 samples).

The preprocessing part is happening in the **pipeline** module – **Pipeline.py – Pipeline class.** Pipeline uses pandas to organise the data: get first 6 classes and remove unwanted columns – keep only the features and the ground-truth. First, *read(filename)* is called to load the data from filename in the memory, more exactly in a DataFrame. Then, *preprocess()* is called to merge some classes and drop some columns (more details about the merging we included in the previous documentation). Finally, *prepare_training* is called to get numerical classes from labels and to split the data in train and test. We chose to use train_test_split because it allows to set a random_state and in this way we can have the same training and testing as we set for the previous implementation using sklearn – by doing this we can compare the models without altering testing and training data sets. All these three steps are incorporated in a single function called *run_all_preliminary_steps(filename)* for simplicity.

2. Decision Trees Module

The **decision_trees** module – **DecisionTree.py – DecisionTree class** covers our implementation for decision trees. The main arguments are **criterion**, **splitter**, **max_depth**, **min_samples_split**, **min_samples_leaf**, **max_thresholds**. For the current version of the implementation criterion can be only entropy, whereas splitter, min_samples_split and min_samples_leaf are not implemented yet. **max_depth** can be set and it represent the maximum depth of the trees and **max_thresholds** is a variable which is present for efficiency reasons and we will describe its use in a few moments.

The core of the training lies in the private method *__build_decision_tree()* called from *fit().* This method will return either:

(1) a decision or internal node – *DecisionNode* which will split the data in two branches based on the best *Question* using a greedy strategy. The greedy strategy consists in splitting the data in such a way that the information gain is maximum. Firstly, the data can be split by a rule that looks like this: **X[feature_index] <= threshold**. Therefore, the algorithm looks to split data based on a specific feature (feature_index) and a threshold that splits in two based on rule R: less or equal than the threshold or greater than the threshold). This information is included in *Question* which is a class that stores the feature_index and the threshold. Question class also stores a match function which mainly implement rule R. The algorithm try each feature and should try each value from input data as a threshold, but for efficiency reasons we included **max_thresholds** which randomly selects a number of **max_thresholds** from all possible thresholds – this new set of thresholds is called trimmed_thresholds. Regarding the information gain, each split divide the data in two sets and our goal is to have order inside a set – we are looking to have a majority of some class such that the splitting is insightful, otherwise we would have samples for all classes which will be a mess. A measure for order is entropy – a value between 0 and 1, 1 meaning high disorder. We compute entropy for both new two subsets and then the information gain is 2 – entropy_left – entropy_right. We chose 2 since maximum entropy is 1, so if both left and right are highly disordered the information gain will be 0 – which is correct, otherwise if entropy would be 0 for left and right (highly ordered), the information gain will be maximum. We chose the splitting with the maximum information gain.

(2) a leaf node – *Leaf*

When we reach maximum depth or we only have samples from a single class, the node become a leaf. Leafs mapped to output labels: we can either take the label class with the maximum occurences (get_max_class) or we can compute probabilities (get_proba_class) and return them.

Regarding the prediction – *predict()*, since the tree is built during the training step, we only search the tree on the best route and when we reach a leaf – we assing the class with the highest probability. We did this recursively (__predict_sample).

Furthermore, *score()* returns the accuracy and *analysis()* returns a dictionary that includes the main performance metrics and the most important model arguments (max_depth, criterion, training_time, max_thresholds).

3. Random Forest Module

The **ensembles** module – **RandomForest.py – RandomForest class** covers our implementation for random forest. The main arguments are **n_estimators, criterion**, **splitter**, **max_depth**, **min_samples_split**, **min_samples_leaf**, **n_jobs**. For the current version of the implementation criterion can be only entropy, whereas splitter, min_samples_split and min_samples_leaf are not implemented yet. **max_depth** can be set and it represent the maximum depth of the trees and **n_jobs** is a variable that represent the number of processes used in the training phase.

In order to train a model, the user should call *fit()* function which is based on *train_estimator(estimator_indexes, X, y, estimators, estimators_features, process=0).* A mention here: *train_estimator* should be a private function, but it is public because of python reasons – Process's target works only with public functions. We will describe now this function which is called only once if n_jobs is None, else n_jobs times on separate processes.

During this function, we train len(estimator_indexes) estimators which are DecisionTrees (either our own implementation or sklearn version). Each estimator is different because the fitting for Random Forest depends on bootstrapping (*__bootstrapping()*) and randomly selecting a subset of features (*__select_subset_of_features()*) – this randomly selector will select a random number of feature between [__min_rate_of_sampling * total_features, total_features].

Regarding the multiprocessing part, firstly we assign each process to train a number of estimators (estimators_indexes indexes) which is computed as total number of estimators divided by number of processes. Each process train an estimator and mark the results in estimators and estimators_features through a Manager class. We need to save estimators_features for the prediction time since for each estimator we have a specific subset of features.

Similarly to Decision Trees, *score()* returns the accuracy and *analysis()* returns a dictionary that includes the main performance metrics and the most important model arguments (n_estimators, criterion, training_time, n_jobs, max_depth).

4. Utils

This module includes two subfiles: (1) our own implementation of performance metrics and (2) a file with our own cross_validation that includes a statistical analysis of results.

The performance metrics are accuracy, prediction, recall, f1_score and (sensitivity which is the same as recall) – (*compute_METRIC(y_pred, y_true, binary=False)*) and we unittested them with sklearn versions as groun_truth (*unittests.test_metrics.py*) - (binary arg is used if we want to treat data as binary or multiclass).

Regarding cross_validation module we use k-fold cross_validation and train k models that will be returned along with the *confident_performance* of the k-fold cross validation training which represents the mean of each metric and the confidence interval computed by the following formula:

$$\alpha = 1.96 \cdot \frac{\sigma}{\sqrt{k}}$$

in method *compute_confidence().*

5. Run

For the running part, we created a simple console (**ConsoleUI)** that can be launched by running python file *run.py.* Three comands can be entered by the user, 0 for exit, 1 for training a random forest model and 2 for training a decision tree model. Both of them allows the user to select an input file and choose specific hyperparmeters and also to choose if the training should use k-fold cross validation and select k. Each step is guided by the console with specific messages.

When the training starts, the console will show some logs for developer and some messages for user. After the training ended, a file with an overview (performances + hyperparams + training time) will be saved and the user will see the name of the file which is going to be saved in the current directory.

For simplicity, we also created an independent .exe file using python module pyinstaller that can be run and follows the same aforementioned scheme – *run.exe.*