CSE454 - HW1 Elifnur Kabalcı 1801042617

1) Pseudocode of Random forest

1) Initilize the # of trees in the forest (#trees) and the # of features to consider at each split (# features)

trees = num_trees
features = num_features

2) for each tree in the forest:
   1) Create a bootstrap sample of the original data.
   2) Grow a decision tree from the bootstrap sample. At each node:
      1) Randomly select (# features) features.
      2) Split the node using the feature that provides the best split according to the objective function, for instance, by maximizing the information gain.
   3) Output the forest of trees for use in prediction.

2)

	Decision Tree	Random forest
Number of Trees	Single tree	Collection of Decision Trees.
feature Selection	The best feature is selected at each node for splitting.	Selects a random subset of features at each node, and the best feature from this subset is used for splitting.
Overfitting	Prone to overfitting, especially when a tree is particularly deep. This is due to the fact that a deep tree will have complex decision rules and thus may fit the noise in the data.	Mitigates this problem by averaging multiple decision trees, leading to a more robust model that generalizes better.
Prediction	Traversing the tree from the root to a leaf node.	Aggregating the predictions of all the trees. (by taking the majority vote for classification or average for regression).
Perform	It depends.	Tend to outperform Decision Trees.

3) Random forest Parameters:

1) n_estimators: # of trees in the forest.
2) criterion: Function that measure the quality of a split. Supported criteria are 'gini' for Gini impurity and 'entropy' for the information gain.
3) max_depth: Maximum depth of the tree.
4) min_samples_split: Minimum # of samples required to split an internal node.
5) min_samples_leaf: Minimum # of samples required to be at a leaf node.
6) min_weight_fraction_leaf: Minimum weighted fraction of the sum total of weights required to be at a leaf node.

7) max_features : # of features to consider when looking for the best split.

8) max_leaf_node: Maximum # of leaf nodes.

9) min_impurity_decrease : A node will be split if this split includes a decrease of the impurity greater than or equal to this value.

10) bootstrap : whether bootstrap samples are used when building trees.

11) oob_score : whether to use out-of-bag samples to estimate the generalization accuracy.

12) n_jobs : # of jobs to run in parallel.

13) random_state : Determines random number generation for both the bootstrapping of the samples used when building trees and the sampling of the features to consider when looking for the best split at each node.

14) verbose : Controls the verbosity when fitting and predicting.

15) warm_start : when set to "True", reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest.

16) class_weight : weights associated with classes in the form {class label: weight}. If not given, all classes are supposed to have weight one.

17) ccp_alpha : Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed.

4) Random Forest:

- Training Time complexity: $O(n \log(n)\, d\, k)$, where n is the # of training examples, d is the # of dimensions of the data, and k is the # of Decision trees.
- Run-time complexity: $O(\text{depth of tree} * k)$
- Space complexity: $O(\text{depth of tree} * k)$

1) K-Nearest Neighbors (KNN)

- Training Time Complexity: $O(knd)$, where k is the # of neighbors, n is the # of training examples and d is the # of dimensions of the data.
- Space complexity: $O(nd)$

2) Logistic Regression

- Training Time Complexity: $O(nd)$, where n is the # of training examples and d is the # of dimensions of the data.
- Space complexity: $O(d)$

3) Decision Tree

- Training Time Complexity: $O(n \log(n)\, d)$, where n is the # of points in the Training set and d is the dimensionality of the data.
- Run-time Complexity: $O(\text{max. depth of tree})$

4) Support Vector Machine (SVM)

- Training Time Complexity: $O(n^2)$, where n is the # of training examples
- Run-time Complexity: $O(kd)$, where k is the # SVM and d is the dimensionality of the data.

5) Naive Bayes

- Training Time Complexity: $O(nd)$, where n is the # of training examples and d is the # of dimensions of the data.
- Run-time Complexity: $O(cd)$, where c is the # of classes and d is the # of dimensions.

=) In terms of time complexity, Logistic Regression and Naive Bayes are generally faster as their time complexity is linear with respect to the # of training examples and dimensions of the data. However, RF can be faster when parallelized across multiple cores for training different Decision Trees.

5) Improving the accuracy

1) Tunning Hyperparameters: n_estimators, max_depth, min_samples_split, min_samples_leaf, max_features
2) Feature Engineering: one_hot encoding for categorical variables, normalization or standardization for numerical variables ...
3) Handling Imbalanced Data: like SMOTE
4) Ensemble Methods: Combining multiple models.
5) Cross-validation: like k-cross validation.

6) Improving the performance

1) Tuning Hyperparameters: n_estimators, max_depth, min_samples_split, min_samples_leaf, max_features
2) Feature Engineering
3) Handling Imbalanced data
4) Ensemble methods
5) Cross validation
6) Parallelization: RF is inherently parallelizable, as each tree is independently constructed. Using multi-core or distributed systems for training can significantly speed up learning and prediction times.

7) Improvement online data

1) Incremental learning: Some implementation of RF support incremental learning, where the model can be updated with new data without needing to retrain from scratch. This is done by updating the trees in the forest with the new data.

2) Sliding window: The model is retrained at regular intervals with the most recent data. This allows the model to adapt to changes over time, but it can be computationally expensive as it involves frequent retraining.

3) Concept Drift Adaptation: In an online setting, the distribution of the data can change over time, a phenomenon known as concept drift. Several strategies can be used to adapt RF to concept drift, such as using a forgetfulness mechanism where old data is given less importance, or using change detection tests to trigger retraining when the data distribution changes significantly.

4) Online Decision Trees: Hoeffding Trees or Extremely fast Decision Tree.

5) Ensemble Methods: Online ensemble methods can be used to combine.

8) Pseudocode:

1) Initialize a supervised learning (eg. SVM, Decision Tree) and an unsupervised learning model (eg. k-means)

2) Apply the unsupervised learning model to the entire dataset (both labeled and unlabeled data) to discover hidden structures or clusters in the data.

3) For each cluster, find the most common label among the labeled instances in that cluster. Assign this label to all the unlabeled instances in the cluster.

4) Train the supervised learning model on the dataset, which now includes the clustering step.

5) Use the trained supervised model to make predictions on new data.

=) This method leverages the strengths of both supervised and unsupervised learning. The unsupervised learning step can uncover hidden patterns in the data that may be useful for the prediction task, while the supervised learning step can make accurate predictions based on the labels.

However, this method also has some limitations. The quality of the labels generated by the unsupervised learning step can significantly impact the performance of the supervised learning model. If the clusters do not correspond well to the classes in the data, the labels may be inaccurate, leading to poor performance. Therefore, it's crucial to choose an appropriate unsupervised learning model and carefully tune it's parameters

3) Pseudocode :

1) Initialize a pre-trained CNN model (eg. VGG16, ResNet, Inception).
2) Remove the last layer (output layer) of the pre-trained model.
3) Add a new output layer that matches the # of classes in the target task.
4) For the pre-trained layers, set the trainable parameter to false. This freezes the weights in these layers so they wont be updated during training.
5) Compile the model with an optimizer and a loss function appropriate for the task (eg. categorical crossentropy for multi-class classification).
6) Train the model on the target data.
7) Evaluate the model on a validation set.
8) If performance on the validation set is not satisfactory, consider fine-tuning the model by unfreezing some of the pre-traned layers and continuing training.
9) Use the trained model to make predictions on new data.

=> Common transfer learning method -> CNN

=> This method leverages the fact that the pre-trained model has already learned useful features from a large dataset, and these features can be useful for the target task, even if it's a different task from the one the model was originally trained on. This is especially useful when the target task has a small amount of data, as it allows us to effectively use a model trained on a large amount of data.