Machine Learning: Assignment 2

Name: Marcel Aguilar Garcia Student Id: 20235620 Class: 2021-CT5143

November 29, 2020

1 Description of the algorithm and design decisions

In this assignment I am presenting the implementation of a Random Forest Classifier.

A **Random Forest Classifier** is an ensemble method that trains several decision trees. In order to calculate the class of an example, the classifier will check the prediction of each tree and return the one that occurs the most.

A Random Forest requires to train several Decision Tree Classifiers. I have implemented the algorithm for **Classification and Regression Trees**. The main characteristic of CART is the fact that it splits nodes into two. The algorithm stops when it does not detect further gain or some pre-set stopping rules are met.

It is reasonable to think that the more different these trees are, the better the Random Forest will perform. This can be done by modifying the traditional CART algorithm to consider a random subset of features to find the best decision in each node (rather than all the features). [1]

Additionally, **bagging** can be applied to use a slightly different set to train each tree. In order to do that, the Random Forest considers a subset of samples with repetition from the train set. In my implementation, the size of this new set will be the same as the original train set unless specified in *Sample Size*.

I have provided a list of hyperparameters that can be used to achieve better performance depending on the problem. As I will be comparing my implementation with scikit-learn, I have used the same naming for these hyperparameters [2]:

- Number of estimators: Total number of trees that are trained by the Random Forest (I will use 100 estimators unless specified.).
- Minimum Sample Split: Minimum number of samples required in a tree node in order to be split
- Bootstrap: If this parameter is set to True then 'bagging' will be applied by the Random Forest
- Sample Size: If Bootstrap is set to True, that's the maximum number of samples that will be taken from the train set
- Maximum Depth: Maximum depth of each tree

Finally, I have allowed 'loss function' to be a parameter of my classifier. I would like to see the performance with a different loss function, Tsallis Entropy, which is a generalisation of Entropy. One advantage of Tsallis Entropy is that it has an additional parameter q that could be used to tune the model [3]:

$$\text{Tsallis Entropy}(\mathbf{y}, \mathbf{q}) := 1 - \sum_{c=1}^{C} p_c(\mathbf{y})^q (\frac{k}{q-1}) \tag{1}$$

where C is the number of classes, $p_c(y)$ is the probability of a specific class in y, q is any real number different than 1 and k is a constant.

See pseudo-code for CART algorithm implemented below:

Algorithm 1: Decision Tree Classifier Algorithm

function decision_tree_classifier(X,y):
1: **if** loss(y) = 0 **then**
2: **return** Leaf(X,y)
3: **end if**
4: feature,value ← **find_best_decision**(X,y)
5: $(X_L, y_L), (X_R, y_R) \leftarrow$ **split_data**$(X, y, feature, value)$
6: left ← **decision_tree_classifier**(X_L, y_L)
7: right ← **decision_tree_classifier**(X_R, y_R)
8: subtree = **NewTree**()
9: *add branch left to subtree*
10: *add branch right to subtree*
11: **return** subtree
12: **end function**

I am not considering hyperparameters in this pseudo-code. *Minimum samples split* and *Maximum depth* can be used as stopping rules in step 1. The aim of the functions used in the algorithm are:

- **find_best_decision**: Iterates through all the features of the tree (or a random sample of features if called from Random Classifier) and through all feature values. Returns the feature and value that minimises the loss function.
  - When referring to *all features values* I want to note that I have done a small improvement in my implementation, instead of considering all feature values I am actually taking the middle points given by them
- **split_data**: Using the feature and value that maximises the gain, splits the train set into two (what would be represented by left and right branch of the tree)

See pseudo-code for Random Forest Classifier algorithm below:

Algorithm 2: Random Forest Classifier Algorithm

function random_forest_classifier(X,y):
1: all_trees = [ ]
2: **for** t in 1,...,N **do**
3: $X_B, y_B \leftarrow$ **get_bootstrap_sample**(X,y)
4: $\mathcal{T}_t \leftarrow$ **decision_tree_classifier**(X_B,y_B)
5: add $\mathcal{T}_t$ to all_trees
6: **end for**
7: **return** all_trees

In order to build a Random Forest Classifier, it's enough to train N different trees using bagging:

- **get_bootstrap_sample**: It returns a random sample with repetition of the train set. In my implementation, this will be done when bootstrap = True. Otherwise, it will take the whole train set.
- N is the number of estimators of the Random Forest. This is a hyperparameter in my implementation and is defaulted to 100.

As I will be comparing my implementation to scikit-learn , I have used the same naming for the hyperparameters and methods. Therefore, my two classes, Random Forest Classifier and Decision Tree Classifier, have methods such as 'predict','predict_proba','fit', etc.

2 Tests and results

To compare my implementation to scikit-learn, I have trained the Random Forest using Gini and Entropy which are the loss functions that scikit-learns provides. On top of that, I have trained my implementation with Tsallis Entropy. I will denote by $\mathcal{C}_G$ (Gini), $\mathcal{C}_E$ (Entropy) and $\mathcal{C}_T$ (Tsallis Entropy) to the classifier given by my implementation and by $\mathcal{C}_{SE}$ (Entropy) and $\mathcal{C}_{SG}$ (Gini) the one from scikit-learn implementation.

The results provided by training each classifier with 10 different random divisions can be seen in the table below:

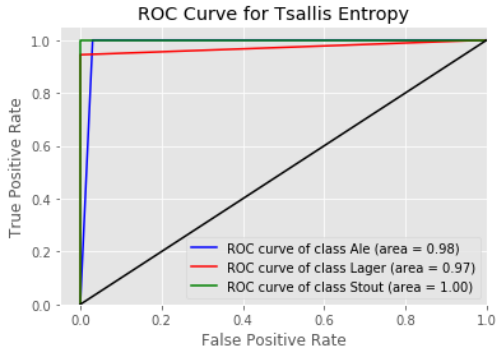
Division	$\mathcal{C}_G$	$\mathcal{C}_E$	$\mathcal{C}_T$	$\mathcal{C}_{SG}$	$\mathcal{C}_{SE}$
Division 1	0.922	0.922	0.961	0.980	0.961
Division 2	0.941	0.922	0.980	0.941	0.941
Division 3	0.941	0.961	0.961	0.961	0.961
Division 4	0.961	0.980	0.961	0.961	0.941
Division 5	1.000	0.980	0.961	0.980	0.980
Division 6	0.922	0.980	0.922	0.961	0.941
Division 7	0.922	0.902	0.902	0.922	0.922
Division 8	0.922	0.961	0.980	0.961	0.980
Division 9	0.941	0.980	0.980	0.941	0.941
Division 10	0.980	0.922	1.000	0.961	0.961
Average	**0.945**	**0.951**	**0.961**	**0.957**	**0.953**

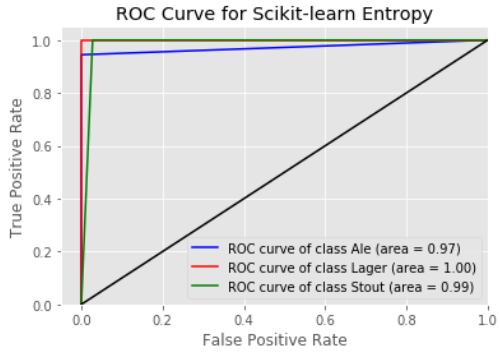
Table 1: Accuracy per division

It's important to note that a Random Forest is a non-deterministic algorithm. Therefore, two random forest with the same parameters and trained with the same set, may have different results. While scikit-learn has a parameter that can be set to build the same classifier repeatedly (random_state), my implementation does not.

2.1 ROC Curve

Now, I will be presenting a ROC Curve for $\mathcal{C}_T$ and $\mathcal{C}_{SE}$ using Division 1. As explained above, the results may be slightly different every time that a classifier is trained. ROC curves are usually used in binary classification problems. In order to extend them to a multi-class case, I will be drawing one curve per class. Each curve represents a class vs the other classes (treated as one). E.g., in the first curve I will consider Ale vs (Lager and Stout).

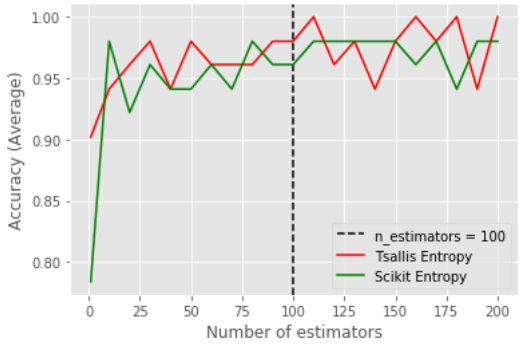




In the case of $\mathcal{C}_T$, the *area under the curve* has the highest value when the class is 'Stout', while for $\mathcal{C}_{SE}$, the highest area is achieved for the class 'Lager'. In both cases, True Positive Rate has a value higher than 0.95 regardless of having a low False Positive Rate.

2.2 Number of estimators

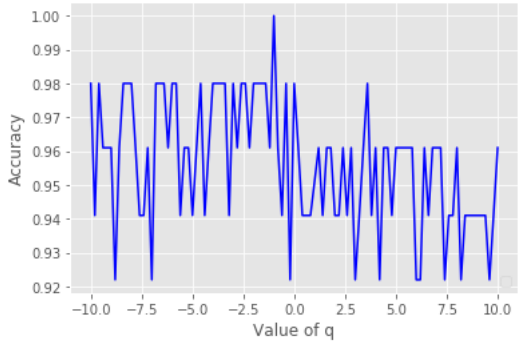
In this assignment I have been using a number of 100 estimators for all classifiers. The following plot shows the averages from Table 1 for $\mathcal{C}_T$ and $\mathcal{C}_{SE}$ in case that a different number is used (ranging from 1 to 200 estimators).



In both cases, having more than 50 estimators, does not seem to guarantee better results and, at the same time, increases the time of execution.

2.3 Optimal q for Tsallis Entropy

Finally, using Division 1, I want to show that q can be used for hyperparameter tuning. In this plot, I show the accuracy for different values of q ranging from -10 to 10.



The optimal q seems to be between -2.5 and 0. In general, $q < 0$ is giving better results.

2.4 Conclusion

From the previous plots, we can say that $\mathcal{C}_T$ and $\mathcal{C}_{SE}$ are showing very similar results and both can be considered good classifiers for this dataset. However, I have found interesting the fact that q can be tuned to improve the accuracy of $\mathcal{C}_T$. It would be nice to see this applied to different datasets.

References

[1] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Taylor & Francis Inc, 2009.

[2] *Scikit-learn: Random Forest Classifier*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`.

[3] *Wikipedia: Tsallis Entropy*. URL: `https://en.wikipedia.org/wiki/Tsallis_entropy`.

Appendix A Random Forest Classifier code:

```
import pandas as pd
import numpy as np
import random
from collections import Counter

def gini(y):
    if len(y)==0: return 0
    _,counts = np.unique(y,return_counts=True)
    probs = counts/counts.sum()
    gini = np.sum(probs*probs)
    return 1-gini

class Stump:

    def __init__(self,y,feature,value,number_of_classes):
        self.y = y
        self.feature = feature
        self.value = value
        self.number_of_classes = number_of_classes

    def count_elements(self):
        count_elements = dict(Counter(self.y))
        return [count_elements.get(n,0) for n in range(0,self.number_of_classes)]


class TreeClassifier:

    def __init__(self,loss_function = gini,min_samples_split = 2,max_depth = 6, count =
        0,forest_flag = 0):

        self.loss_function = loss_function
        self.max_depth   = max_depth
        self.count       = count
        self.forest_flag = forest_flag
        self.min_samples_split = min_samples_split
        self.features    = None
        self.number_of_classes = None
        self.tree        = None

    def fit(self,X,y,count = 0):

        if(count == 0):
            self.features = range(X.shape[1])
            self.number_of_classes = np.unique(y).shape[0]

        if ((np.abs(self.loss_function(y)) < 1e-5) or (count == self.max_depth) or (len(y) <
            self.min_samples_split)):
            return Stump(y,None,None,self.number_of_classes)

        best_feature,best_value,left_idxs,right_idxs = self.find_best_decision(X,y)

        if len(left_idxs) == 0:
            return Stump(y[right_idxs],None,None,self.number_of_classes)

        elif len(right_idxs) == 0:
            return Stump(y[left_idxs],None,None,self.number_of_classes)
```

```
        else:
            stump = Stump(y,best_feature,best_value,self.number_of_classes)
            sub_tree = {stump:[]}
            left_branch = self.fit(X[left_idxs,:],y[left_idxs],count+1)
            right_branch = self.fit(X[right_idxs,:],y[right_idxs],count+1)
            sub_tree[stump].append(left_branch)
            sub_tree[stump].append(right_branch)
            self.tree = sub_tree
            return sub_tree

    def predict(self,X):

        prediction = []

        if self.tree == None:
            print('This tree has not been fit yet')
            return -1

        else:
            for example in X:
                prediction.append(self.predict_example(example,self.tree))
            return np.array(prediction)

    def predict_example(self,example,tree):

        stump = list(tree.keys())[0]
        feature_name = stump.feature
        value  = stump.value

        if example[feature_name] <= value:
            next_branch = tree[stump][0]
        else:
            next_branch = tree[stump][1]

        if not isinstance(next_branch, dict):
            classes_stump = next_branch.count_elements(self)
            return stump.index(max(classes_stump))
        else:
            return self.predict_example(example, next_branch)


    def predict_proba(self,X):
        prediction = []

        if self.tree == None:
            print('This tree has not been fit yet')
            return -1

        else:
            for row in X:
                prediction.append(self.predict_example_proba(row,self.tree))
            return np.array(prediction)

    def predict_example_proba(self,example,tree):

        stump = list(tree.keys())[0]
        feature_name = stump.feature
        value  = stump.value
```

```
    if example[feature_name] <= value:
        next_branch = tree[stump][0]
    else:
        next_branch = tree[stump][1]

    if not isinstance(next_branch, dict):
        classes_stump = np.array(next_branch.count_elements())
        return classes_stump/np.sum(classes_stump)
    else:
        return self.predict_example_proba(example, next_branch)


def split_exemples(self,X,feature,value):
    return {'L':np.where(X[:,feature] <= value)[0],'R':np.where(X[:,feature] > value)[0]}

def get_values_splits(self,X):

    feature_values = np.sort(np.unique(X))

    if len(feature_values) == 1:
        return [feature_values[0]]
    else:
        return [(feature_values[n]+feature_values[n+1])/2 for n in range(len(feature_values)-1)]

def find_best_split(self,X,y,feature):

    values_splits = self.get_values_splits(X[:,feature])
    total_values = X.shape[0]
    minimum_loss = np.inf

    for value in values_splits:

        split_examples = self.split_exemples(X,feature,value)
        left_idxs = split_examples['L']
        right_idxs = split_examples['R']
        Lprob = len(left_idxs)/total_values
        Rprob = len(right_idxs)/total_values
        Lloss = self.loss_function(y[left_idxs])
        Rloss = self.loss_function(y[right_idxs])

        loss_split = Lprob*Lloss+Rprob*Rloss

        if loss_split < minimum_loss:
            minimum_loss = loss_split
            results = [value,minimum_loss,left_idxs,right_idxs]

    return results

def find_best_decision(self,X,y):

    lowest_loss = np.inf

    if self.forest_flag == 1:
        features = np.random.choice(self.features,int(round(np.sqrt(X.shape[1]))),replace=False)

    for feature in features:

        best_value,minimum_loss,left_idxs,right_idxs = self.find_best_split(X,y,feature)
```

```
            if  minimum_loss < lowest_loss:
                lowest_loss = minimum_loss
                result = [feature,best_value,left_idxs,right_idxs]

        return result

class RandomForestClassifier:

    def __init__(self,n_estimators = 100, sample_size = None,loss_function = gini,
         min_samples_split = 2,verbose = 0,bootstrap = True):
        self.n_estimators = n_estimators
        self.sample_size = sample_size
        self.min_samples_split = min_samples_split
        self.trees = None
        self.loss_function = loss_function
        self.verbose = verbose
        self.bootstrap = bootstrap

    def predict(self, X):
        probs = self.predict_proba(X)
        return np.argmax(probs, axis=1)

    def predict_proba(self, X):
        return np.round(np.mean([t.predict_proba(X) for t in self.trees], axis=0),2)


    def fit(self, X,y):

        if(self.verbose == 1):
            print(f'Tree number {k+1}')

        if self.sample_size == None:
            self.sample_size = X.shape[0]

        self.trees =
            [TreeClassifier(self.loss_function,min_samples_split=self.min_samples_split,max_depth
            = -1,forest_flag = 1) for i in range(self.n_estimators)]

        for rd,t in enumerate(self.trees):
            if self.bootstrap:
                rnd_idxs = random.choices(range(X.shape[0]), k=self.sample_size)
            else:
                rnd_idxs = range(X.shape[0])

            t.fit(X[rnd_idxs],y[rnd_idxs])
```