Design and Implementation of Iterative Dichotomiser 3 (ID3) Decision Tree Classifier with Python

Sami Alperen Akgun
sami.alperen.akgun@gmail.com

I. ID3 Based on Information Gain

A. Implementation

Iterative Dichotomiser 3 (ID3) decision tree classifier based on maximizing information gain (IG) was implemented for this question in Python programming language using numpy library. The implemented functions $BuildingDT$ and $BuildingDT_{cont}$ for data with discrete attributes and data with continuous attributes respectively can be seen below:

```
def BuildingDT(input_X,input_Y,feature_X_values,feature_Y_values,Attribute,mc_label):
    """
        This function creates an ID3 decision tree
        Inputs: input_X --> np array contains data with features/attributes
                input_Y --> np array contains output labels
                feature_X_values --> list composed of possible values of features
                feature_Y_values --> list composed of possible values of out labels
                Attribute --> selected attribute/feature number(s) (column index)
                              It is a list composed of column indices
                mc_label --> most common label in the original data Y
        Output: Decision Tree DT (class)
    """
    # If there is no output label
    if input_Y.shape[0] == 0:
        return DT(LeafValue=mc_label)

    children = [] #children nodes from the current selected node
    mcv = most_common_value(input_Y,feature_Y_values) # most common value of given input Y

    # If all labels of input_Y is the same or Attribute is empty
    all_values_same_boolean = check_all_values_same(input_Y,feature_Y_values)
    if all_values_same_boolean or Attribute == []:
        return DT(LeafValue=mcv)
    else:
        best_IG = 0
        best_set_X = []
        best_set_Y = []
        best_Attribute = 0
        current_entrophy = entrophy(input_Y,feature_Y_values)
        for index in Attribute:
            # index is the column index --> decides which attribute to use
            divided_X, divided_Y = divide_data(input_X,input_Y,index,feature_X_values)

            for j in range(len(divided_Y)): #iterate through each possible feature value
                # For tic-tac-toe --> divided_Y contains subsets for "x", "o" and "b"
                current_IG = information_gain(divided_Y[j],feature_Y_values,current_entrophy)
                if current_IG >= best_IG:
                    best_IG = current_IG
                    best_Attribute = index
                    best_feature_ind = j


        # Now best attribute is selected --> divide data using that attribute
        # best_Attribute --> gives the column number of best attribute
        # best_feature_ind --> gives which feature value has the most IG, i.e. "x", "o" or "b" in tic-tac
        best_set_X, best_set_Y = divide_data(input_X,input_Y,best_Attribute,feature_X_values)

        # Remove the current attribute from the attribute list to
        # create remaning attributes --> no need to use same attribute over and
        # over again! --> If it is needed, it is gonna show up in the recursion anyway
        remaning_Attributes = []
        for index in Attribute:
```

```
            if index != best_Attribute:
                remaning_Attributes.append(index)


        for i in range(len(best_set_X)):
            children.append( BuildingDT(best_set_X[i],best_set_Y[i],feature_X_values,feature_Y_values,
                                        remaning_Attributes,mcv) )

        return DT(FeatureNum=best_Attribute,Children=children)
```

```
def BuildingDT_cont(input_X,input_Y,feature_X_values,feature_Y_values,Attribute,mc_label):
    """
        This function creates an ID3 decision tree
        Inputs: input_X --> np array contains data with features/attributes
                input_Y --> np array contains output labels
                feature_X_values --> list composed of possible values of features
                feature_Y_values --> list composed of possible values of out labels
                Attribute --> selected attribute/feature number(s) (column index)
                              It is a list composed of column indices
                mc_label --> most common label in the original data Y
        Output: Decision Tree DT (class)
    """
    # If there is no output label
    if input_Y.shape[0] == 0:
        return DT_cont(LeafValue=mc_label)

    left = [] # contains values less or equal than threshold
    right = [] # higher than threshold
    mcv = most_common_value(input_Y,feature_Y_values) # most common value of given input Y

    # If all labels of input_Y is the same or Attribute is empty
    all_values_same_boolean = check_all_values_same(input_Y,feature_Y_values)
    if all_values_same_boolean or Attribute == []:
        return DT_cont(LeafValue=mcv)
    else:
        best_IG = 0
        best_set_X = []
        best_set_Y = []
        best_Attribute = 0
        best_threshold = 0.0
        current_entrophy = entrophy(input_Y,feature_Y_values)
        for index in Attribute:
            # index is the column index --> decides which attribute to use
            all_thresholds = return_all_thresholds(input_X,input_Y,index)
            for threshold in all_thresholds:
                # Get divided data for each threshold for a given attribute = index
                divided_X, divided_Y = divide_data(input_X,input_Y,index,feature_X_values,threshold)

                for j in range(len(divided_Y)): #iterate through each possible feature value
                    # For wine.data --> divided_Y contains subsets for lower then threshold
                    # and higher than the threshold
                    current_IG = information_gain(divided_Y[j],feature_Y_values,current_entrophy)
                    if current_IG >= best_IG:
                        best_IG = current_IG
                        best_Attribute = index
                        best_feature_ind = j
                        best_threshold = threshold


        # Now best attribute and threshold are selected --> divide data using that attribute and thresh
        # best_Attribute --> gives the column number of best attribute
        # best threshold --> gives the best threshold value which gives max IG
        # best_feature_ind --> gives which feature value has the most IG, i.e. "x", "o" or "b" in tic-tac
        best_set_X, best_set_Y = divide_data(input_X,input_Y,best_Attribute,feature_X_values,best_threshold
    )

        # Remove the current attribute from the attribute list to
        # create remaning attributes --> no need to use same attribute over and
        # over again! --> If it is needed, it is gonna show up in the recursion anyway
        remaning_Attributes = []
        for index in Attribute:
            if index != best_Attribute:
                remaning_Attributes.append(index)

```

```
        # Left contains values less or equal than threshold
        # Right contains values more than threshold
        # For more detail, check divide data function

        return DT_cont(Left=BuildingDT_cont(best_set_X[0],best_set_Y[0],feature_X_values,feature_Y_values,
                        remaning_Attributes,mcv),
                       Right=BuildingDT_cont(best_set_X[1],best_set_Y[1],feature_X_values,feature_Y_values,
                        remaning_Attributes,mcv),
                       FeatureNum=best_Attribute,ThresholdNum=best_threshold
                       )
```

B. Training and Performance

10-times-10-fold cross validation technique was employed to divide the data into training and test set. Decision trees were created using training data and accuracy of them were calculated using test data.

Here are the obtained results for tic-tac-toe endgame dataset:

```
Mean of the accuracy:  0.78875 --> 78%
Var of the accuracy:  2.7769225320698284e-07
Best accuracy:  0.8854166666666666 --> 89%
```

Here are the obtained results for wine dataset:

```
Mean of the accuracy:  0.7816666666666667 --> 78%
Var of the accuracy:  1.4005687014174652e-05
Best accuracy:  0.944444444444444 --> 94%
```

Finally, confusion matrices were calculated for the tree which gives the best accuracy among 10-times-10-fold cross validation. It can be seen in Table I, where rows shows actual classes and columns show predicted classes.

TABLE I
CONFUSION MATRICES

	Predicted Numbers	
	Positive	Negative
Positive	60	5
Negative	6	25

(a) Confusion Matrix for tic-tac-toe endgame Dataset

	Predicted Numbers		
	Class "1"	Class "2"	Class "3"
Class "1"	6	0	0
Class "2"	0	7	0
Class "3"	0	1	4

(b) Confusion Matrix for wine Dataset

Average mean of accuries for both dataset are almost the same, but variance in wine dataset is lower than tic-tac-toe dataset, which shows that decision tree trained for wine dataset is more successful than the other one since result of 10-times-10-fold cross validation don't change too much. If we look at confusion matrices, we can infer that classifier for tic-tac-toe dataset has balanced errors, i.e. the wrong prediction numbers are almost the same for positive and negative label case. On the other hand, confusion matrix for wine dataset clearly shows that trained decision tree is almost perfect and it predicted only one example wrong, which was normally belong to class 3, but tree predicted as class 2. One last thing to note that, since number of test samples are quite low for wine dataset, even having one wrong prediction reduces the accuracy a lot in a percentage wise.

II. ID3 BASED ON GAIN RATIO

A. Implementation

Same procedure was executed once again, but this time decision trees were created based on gain ratio (GR) instead of information gain (IG). One can see implemented functions $BuildingDT_{GR}$ and $BuildingDT_{cont}GR$ for data with discrete attributes and data with continious attributes below:

```
def BuildingDT_GR(input_X,input_Y,feature_X_values,feature_Y_values,Attribute,mc_label):
    """
        This function creates an ID3 decision tree based on maximizing gain ratio (GR)
        Inputs: input_X --> np array contains data with features/attributes
                input_Y --> np array contains output labels
                feature_X_values --> list composed of possible values of features
                feature_Y_values --> list composed of possible values of out labels
                Attribute --> selected attribute/feature number(s) (column index)
                              It is a list composed of column indices
                mc_label --> most common label in the original data Y
        Output: Decision Tree DT (class)
    """
```

```
    # If there is no output label
    if input_Y.shape[0] == 0:
        return DT(LeafValue=mc_label)

    children = [] #children nodes from the current selected node
    mcv = most_common_value(input_Y,feature_Y_values) # most common value of given input Y

    # If all labels of input_Y is the same or Attribute is empty
    all_values_same_boolean = check_all_values_same(input_Y,feature_Y_values)
    if all_values_same_boolean or Attribute == []:
        return DT(LeafValue=mcv)
    else:
        best_GR = 0 #best gain ratio
        best_set_X = []
        best_set_Y = []
        best_Attribute = 0
        current_entrophy = entrophy(input_Y,feature_Y_values)
        for index in Attribute:
            # index is the column index --> decides which attribute to use
            divided_X, divided_Y = divide_data(input_X,input_Y,index,feature_X_values)

            for j in range(len(divided_Y)): #iterate through each possible feature value
                # For tic-tac-toe --> divided_Y contains subsets for "x", "o" and "b"
                current_IG = information_gain(divided_Y[j],feature_Y_values,current_entrophy)
                current_GR = gain_ratio(divided_Y[j],feature_Y_values,current_IG)
                if current_GR >= best_GR:
                    best_GR = current_GR
                    best_Attribute = index
                    best_feature_ind = j


        # Now best attribute is selected --> divide data using that attribute
        # best_Attribute --> gives the column number of best attribute
        # best_feature_ind --> gives which feature value has the most IG, i.e. "x", "o" or "b" in tic-tac
        best_set_X, best_set_Y = divide_data(input_X,input_Y,best_Attribute,feature_X_values)

        # Remove the current attribute from the attribute list to
        # create remaning attributes --> no need to use same attribute over and
        # over again! --> If it is needed, it is gonna show up in the recursion anyway
        remaning_Attributes = []
        for index in Attribute:
            if index != best_Attribute:
                remaning_Attributes.append(index)


        for i in range(len(best_set_X)):
            children.append( BuildingDT(best_set_X[i],best_set_Y[i],feature_X_values,feature_Y_values,
                                        remaning_Attributes,mcv) )

        return DT(FeatureNum=best_Attribute,Children=children)
```

```
def BuildingDT_cont_GR(input_X,input_Y,feature_X_values,feature_Y_values,Attribute,mc_label):
    """
        This function creates an ID3 decision tree based on maximizing gain ratio (GR)
        Inputs: input_X --> np array contains data with features/attributes
                input_Y --> np array contains output labels
                feature_X_values --> list composed of possible values of features
                feature_Y_values --> list composed of possible values of out labels
                Attribute --> selected attribute/feature number(s) (column index)
                              It is a list composed of column indices
                mc_label --> most common label in the original data Y
        Output: Decision Tree DT (class)
    """
    # If there is no output label
    if input_Y.shape[0] == 0:
        return DT_cont(LeafValue=mc_label)

    left = [] # contains values less or equal than threshold
    right = [] # higher than threshold
    mcv = most_common_value(input_Y,feature_Y_values) # most common value of given input Y

    # If all labels of input_Y is the same or Attribute is empty
    all_values_same_boolean = check_all_values_same(input_Y,feature_Y_values)
    if all_values_same_boolean or Attribute == []:
```

```
        return DT_cont(LeafValue=mcv)
    else:
        best_GR = 0
        best_set_X = []
        best_set_Y = []
        best_Attribute = 0
        best_threshold = 0.0
        current_entrophy = entrophy(input_Y,feature_Y_values)
        for index in Attribute:
            # index is the column index --> decides which attribute to use
            all_thresholds = return_all_thresholds(input_X,input_Y,index)
            for threshold in all_thresholds:
                # Get divided data for each threshold for a given attribute = index
                divided_X, divided_Y = divide_data(input_X,input_Y,index,feature_X_values,threshold)

                for j in range(len(divided_Y)): #iterate through each possible feature value
                    # For wine.data --> divided_Y contains subsets for lower then threshold
                    # and higher than the threshold
                    current_IG = information_gain(divided_Y[j],feature_Y_values,current_entrophy)
                    current_GR = gain_ratio(divided_Y[j],feature_Y_values,current_IG)
                    if current_GR >= best_GR:
                        best_GR = current_GR
                        best_Attribute = index
                        best_feature_ind = j
                        best_threshold = threshold


        # Now best attribute and threshold are selected --> divide data using that attribute and thresh
        # best_Attribute --> gives the column number of best attribute
        # best threshold --> gives the best threshold value which gives max IG
        # best_feature_ind --> gives which feature value has the most IG, i.e. "x", "o" or "b" in tic-tac
        best_set_X, best_set_Y = divide_data(input_X,input_Y,best_Attribute,feature_X_values,best_threshold
    )

        # Remove the current attribute from the attribute list to
        # create remaning attributes --> no need to use same attribute over and
        # over again! --> If it is needed, it is gonna show up in the recursion anyway
        remaning_Attributes = []
        for index in Attribute:
            if index != best_Attribute:
                remaning_Attributes.append(index)

        # Left contains values less or equal than threshold
        # Right contains values more than threshold
        # For more detail, check divide data function
        #left.append( BuildingDT_cont(best_set_X[0],best_set_Y[0],feature_X_values,feature_Y_values,
                                              #remaning_Attributes,mcv) )
        #right.append( BuildingDT_cont(best_set_X[1],best_set_Y[1],feature_X_values,feature_Y_values,
                                              #remaning_Attributes,mcv) )
        return DT_cont(Left=BuildingDT_cont(best_set_X[0],best_set_Y[0],feature_X_values,feature_Y_values,
                        remaning_Attributes,mcv),
                       Right=BuildingDT_cont(best_set_X[1],best_set_Y[1],feature_X_values,feature_Y_values,
                        remaning_Attributes,mcv),
                       FeatureNum=best_Attribute,ThresholdNum=best_threshold
                       )
```

B. Training and Performance

Again 10-times-10-fold cross validation was executed to evaluate the implemented decision trees.
Here are the obtained results for tic-tac-toe endgame dataset:

```
Mean of the accuracy:  0.7888541666666666 --> 79%
Var of the accuracy:  6.982629681810922e-07
Best accuracy:  0.8958333333333334 --> 90%
```

Here are the obtained results for wine dataset:

```
Mean of the accuracy:  0.7822222222222223 --> 78%
Var of the accuracy:  1.2356500533455256e-05
Best accuracy:  1.0 --> 100%
```

Finally, confusion matrices were calculated for the tree which gives the best accuracy among 10-times-10-fold cross validation. It can be seen in Table II

TABLE II
CONFUSION MATRICES

Predicted Numbers	
Positive	Negative
59	5
5	27

(a) Confusion Matrix for tic-tac-toe endgame Dataset

Predicted Numbers		
Class "1"	Class "2"	Class "3"
6	0	0
0	8	0
0	0	4

(b) Confusion Matrix for wine Dataset

When the decision tree was built based on gain ratio instead of information gain, the performance of classifier for tic-tac-toe dataset was not changed significantly although there is a significant change for wine dataset. This behaviour is quite expected since generally gain ratio was used when a given input data has so many attribute values which force classifier based on information gain to choose that attribute. For tic-tac-toe dataset, all attributes has only three values: "x", "o" and "b", so building classifier based on gain ratio doesn't create any difference. This is also true for wine dataset, whose attributes are continious and they are converted to low and high than a certain threshold during the process, so they have only two different values. The observed difference was caused from the number of test samples for wine dataset. It is quite low, so small changes create big differences as percentage wise. In summary, obtained results are similar for both IG and GR case, since given attributes of the data don't have too many values.

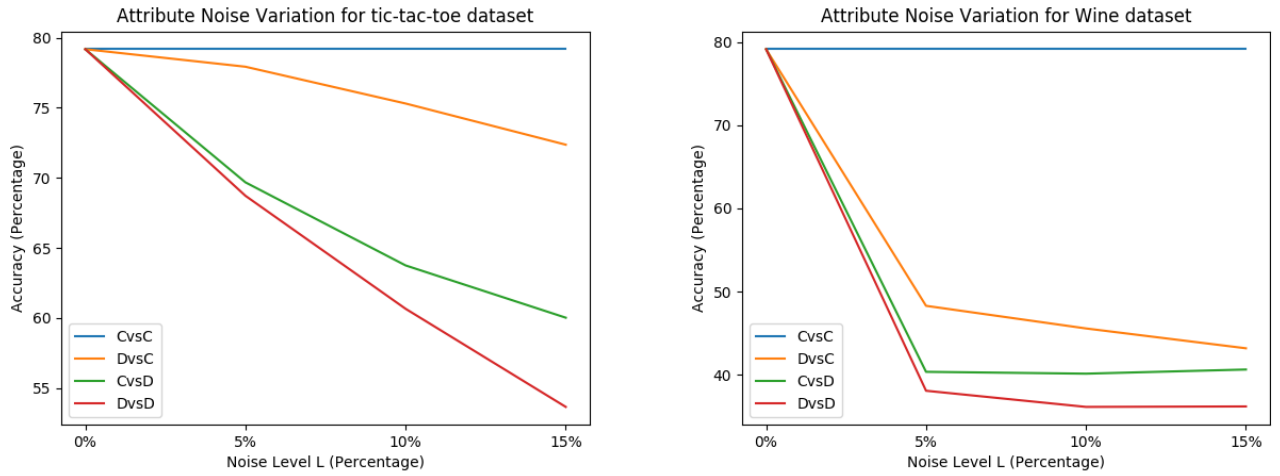
III. UNDERSTANDING ATTRIBUTE AND CLASS NOISE ON THE PERFORMANCE OF ID3 CLASSIFIER

A. Analysis the Effect of Attribute Noise

Plots in Figure 1 show the classification accuracy respect to different attribute noises with the values of (5%, 10%, and 15%) for tic-tac-toe endgame and wine dataset. From these plots, we can conclude that corrupting test set is much more dangerous than corrupting training set since mean accuracy of 10-times-10-fold cross validation drops abruptly whenever test set is corrupted. Therefore, accuracy values are in the following order for both of the datasets:

```
Clean Training Set vs Clean Test Set (CvsC) > Dirty Training Set vs Clean Test Set (DvsC) > Clean Training
    Set vs Dirty Test Set (CvsD) > Dirty Training Set vs Dirty Test Set (DvsD)
```

The difference between datasets is reduction amount in accuracy is lower as noise level increases for wine dataset. It is not surprising since test set size is low for wine dataset, so corrupting a few samples make huge impact. One last note about the plots is that sometimes the trends in the curves are not as expected like CvsD case for wine dataset. The reason behind is the combination of randomness of the added noise and low test set size, i.e. sometimes flipping values randomly might not create too much change in the test set. For example, when test set has 8 samples, flipping randomly the values of 8 samples might create a new test set whose values are the same as old one like 6 out of 8 values might stay the same.



(a) tic-tac-toe dataset

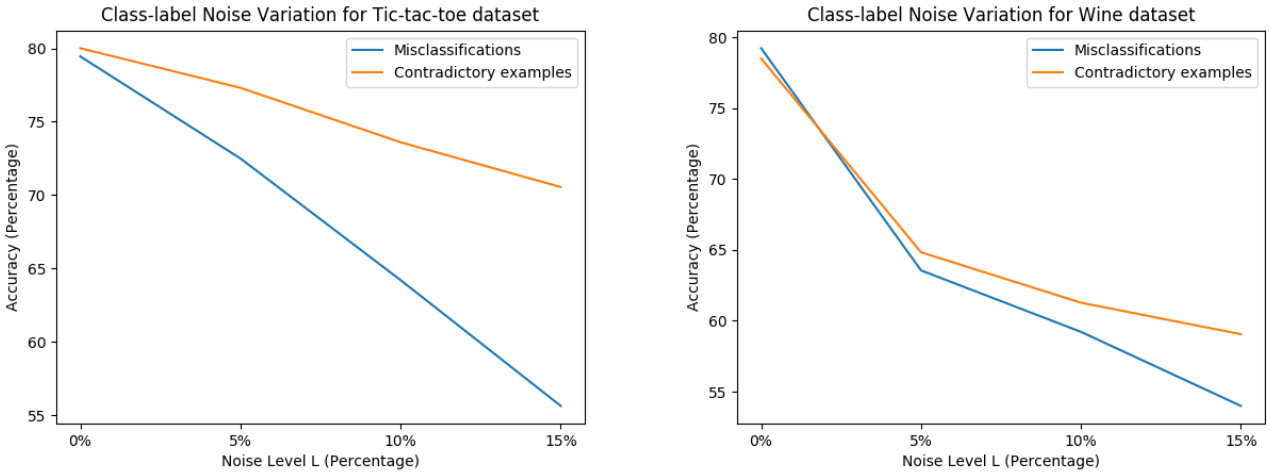
(b) wine dataset

Fig. 1. Variation of Attribute Noise for Given Datasets

B. Analysis the Effect of Class-label Noise

Plots in Figure 2 show the classification accuracy respect to different class-label noises with the values of (5%, 10%, and 15%) for tic-tac-toe endgame and wine dataset. From these plots, we can infer that misclassifications in the training data affects the accuracy more than contradictory examples in the training data for both of the datasets. This makes sense since for contradictory examples, decision tree has both of the labels, so somehow it can self-correct itself or choose to not use these examples since same they point out different labels with same features. However, decision tree performs quite bad when there is misclassifications in the training data since it is not easy to detect misclasffications from clear training data.

The difference between these two datasets are the rate of reduction in accuracy. Reductions are much more dramatic in wine dataset due to small test sample size comparing to tic-tac-toe dataset.



(a) tic-tac-toe dataset

(b) wine dataset

Fig. 2. Variation of Class-label Noise for Given Datasets

C. Prunning Variable and ID3

When there is not any prunning variable, having noise in the training data creates overfitting problem since ID3 tries to grow each branch of the decision tree in such a way that it can perfectly classify the data. When there is a noise in training data, the tree is erroneously fitting which creates overfitting problem or/and huge decision trees. This becomes quite crucial especially when the training data is too small like in our case with wine dataset. That's why post=prunning methods has been widely used in practice to overcome this problem.

D. Attribute Noise vs Class Noise

Class noise is more harmful than attribute noises since attribute noise changes the values of features while class noise changes directly the training labels. As a result, class noise also changes the balance of labels in the training data and it creates a class imbalance. For example, if you have a minority in your training data less say with label "1", if class noise changes these labels to majority, then you will lose quite precious information related to class "1" since there is limited number of examples. Therefore, in the end accuracy will drop significantly when this happens. It is problematic from decision tree perspective as well since it can change most common value in labels which will directly affect the tree structure negatively. Hence, class noise is usually more harmful than attribute noise. Attribute noise still can be managed to overcome if we have enough training samples since at the end decision tree itself is learning from given training data to map features with labels.