Ensemble Learning Project

Yago Bardi Vale, Caterina Conz, Nevina Dalal, Vivian Koutroumani

Contents

In today's oral Défense we are going to cover the following aspects:

1. Classification of cyberbullying tweets using various ensemble methods
    - Exploratory Data Analysis
    - Data Pre-processing & Feature Engineering
    - Binary Classification to understand which tweets are considered harmful
    - Multiclass classification of tweets
    - Evaluation and comparison
2. Process of building a decision tree from scratch using classical libraries
    - Datasets used for each type of decision tree
    - Comparison to sklearn

01
Classification of Cyberbullying tweets

Problem Definition

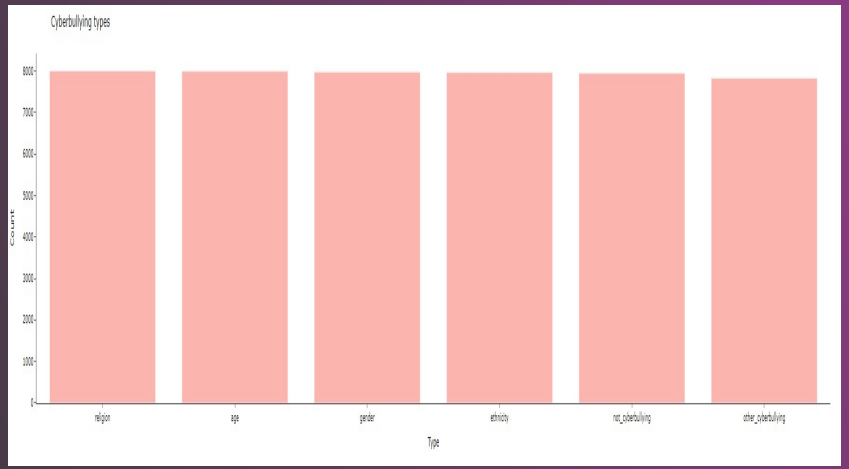
Bullying that occurs through the use of digital technologies such as cell phones, computers, and tablets is known as cyberbullying. sending and posting harmful, negative, inaccurate, or derogatory information about someone is included. 36.5 percent of pupils have experienced cyberbullying, and 87 percent have witnessed it.

Our model is trained on a dataset of tweets and aims to predict whether a tweet is related to cyberbullying, and if so, what kind of cyberbullying : religion, gender, age, ethnicity, other.

Exploratory Data Analysis

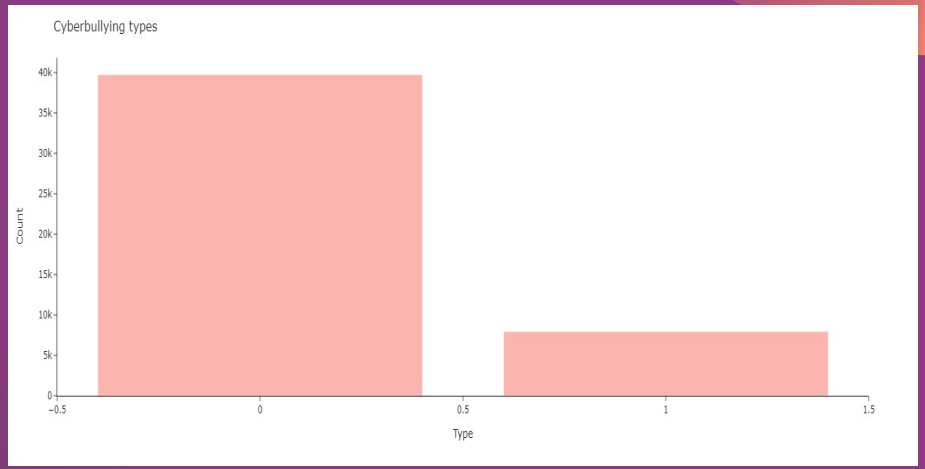
Overview of the Classes

Multiclass Classification



Data is evenly balanced across classes

Binary Classification

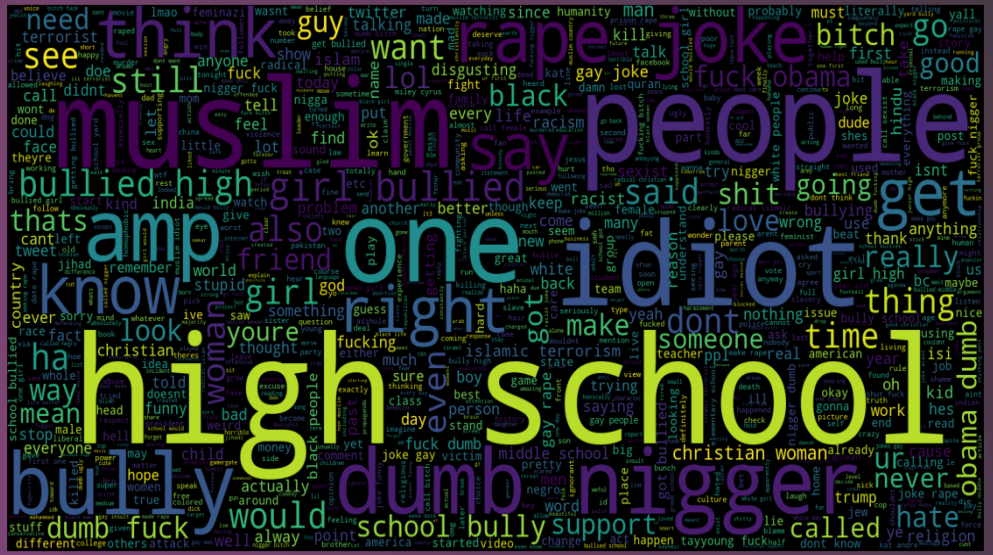


There is severe imbalance in favour of cyberbullying

Frequently Used Words

Many problematic words did get highlighted but we could not associate them to any particular class of cyber bullying

To classify better we decided to look at frequency per class



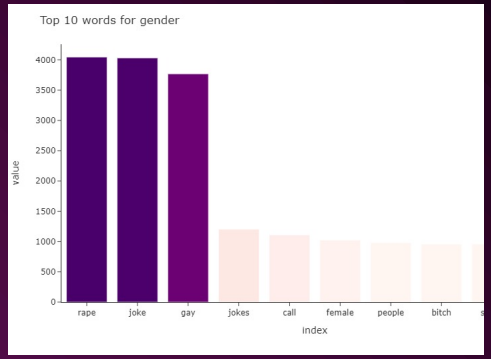
Top Words Per Category

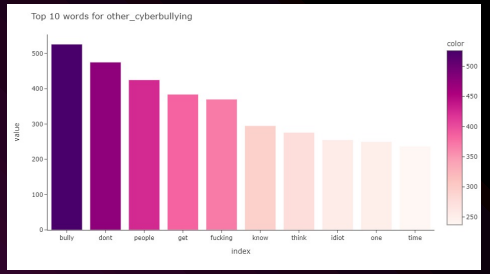
Words relevant to each type of cyber bullying now seems more relatable

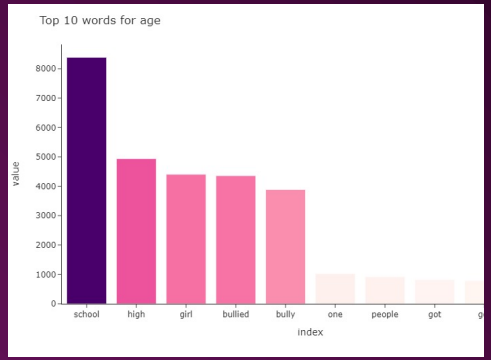
We would expect that the classifier would also give more importance to these words while classifying

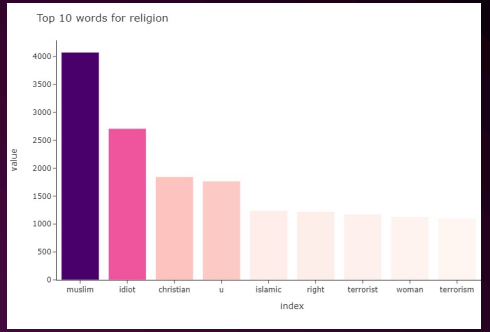
However, unigrams alone do not hold a lot of meaning. For Ex: high and school are two words that would ideally be used together

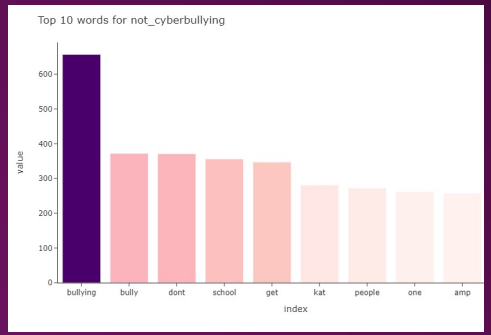
We hence decided to analyze bigrams per category

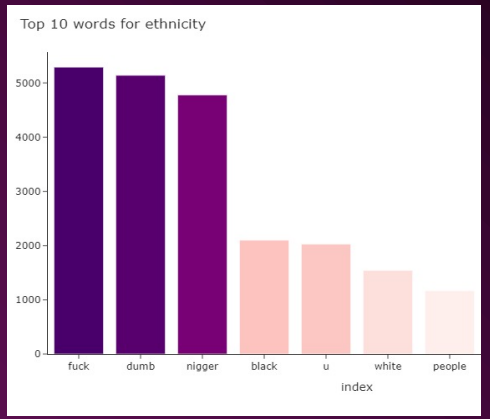


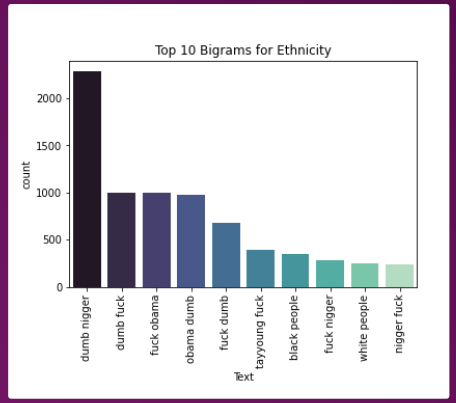


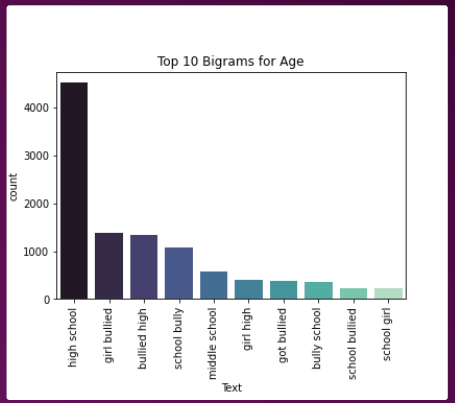


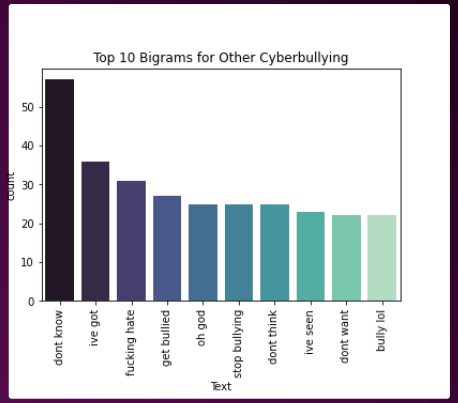


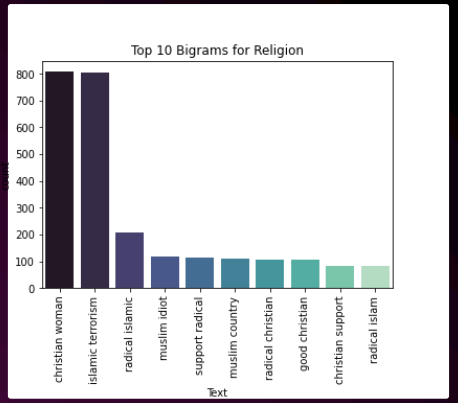


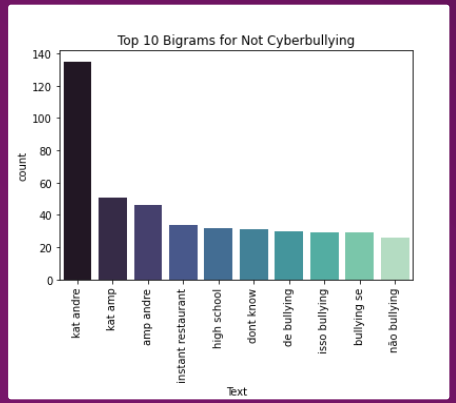


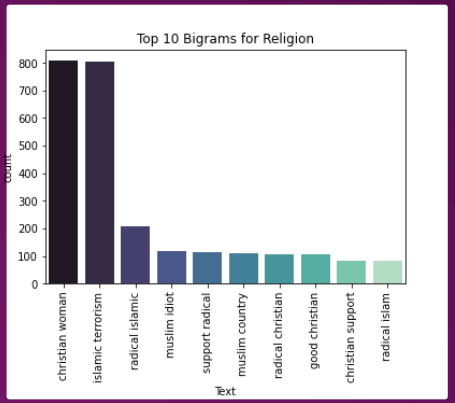












Top 10 Bigrams Per Category

- The problematic words seem to make more sense now

Feature Engineering

Feature Engineering – Style features

Nº of characters	Non ASCII	Nº of capital letters
54	0	1
48	0	1
21	0	0

Anon_32
@anon_32

The same reason you gave the bully your lunch money back in school

Pre Processing

01

Tokenization

[same, reason, gave, bully, lunch, money, back, school]

02

Lemmatization

[same, reason, give, bully, lunch, money, back, school]

Feature Engineering

01

Bag of Words

The occurrence of each word is used as feature
Keep only the top 100 most frequent words

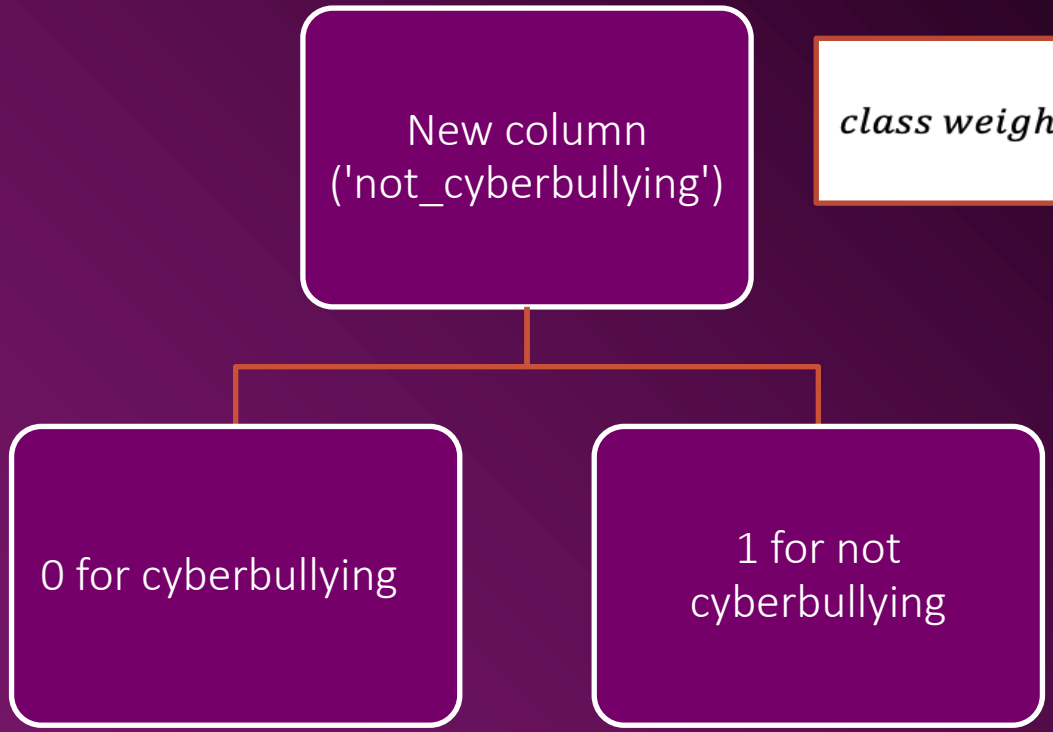
02

TF-IDF MATRIX

Term frequency – inverse document frequency: intends to capture how important a word is to a specific document

Binary Classification

What makes a tweet harmful?



$$class\ weights = \frac{Number\ of\ 0s}{Number\ od\ 1s} = 5$$

Handling Class Imbalance

Oversampling the small class
(not cyber bullying)

Setting the class weights parameter for each model

Undersampling the big class (cyberbullyng)

Experimenting with the threshold that classifies each tweet

Binary Classification Results

Evaluated based on f1-score and the recall of class 1 (the smaller class) and the overall accuracy score.

- f1-score: balancing techniques didn't improve the score, best score from RF on original classes. It is better for imbalanced data
- Recall: overbalancing and underbalancing improved a lot this score, with XGBoost on underbalanced classes giving 98%.

F1-score	RF	DT	CatBoost	AdBoost	XGBoost
Imbalanced	0.60	0.45	0.5	0.44	0/39
Overbalanced	0.59	0.49	0.59	0.59	0.59
Underbalanced	0.58	-.55	0.58	0.59	0.58
Class weight	0.60	0.47	0.59	N/A	0.39
Threshold	0.60	0.45	0.59	0.44	0.39

Recall	RF	DT	CatBoost	AdBoost	XGBoost
Imbalanced	0.89	0.44	0.39	0.33	0/29
Overbalanced	0.92	0.51	0.95	0.95	0.97
Underbalanced	0.96	0.77	0.96	0.94	0.98
Class weight	0.89	0.51	0.94	N/A	0.27
Threshold	0.89	0.44	0.94	0.33	0.27

Binary Classification Results

Overall accuracy: Decreasing as the above scores increasing (76%-87%)

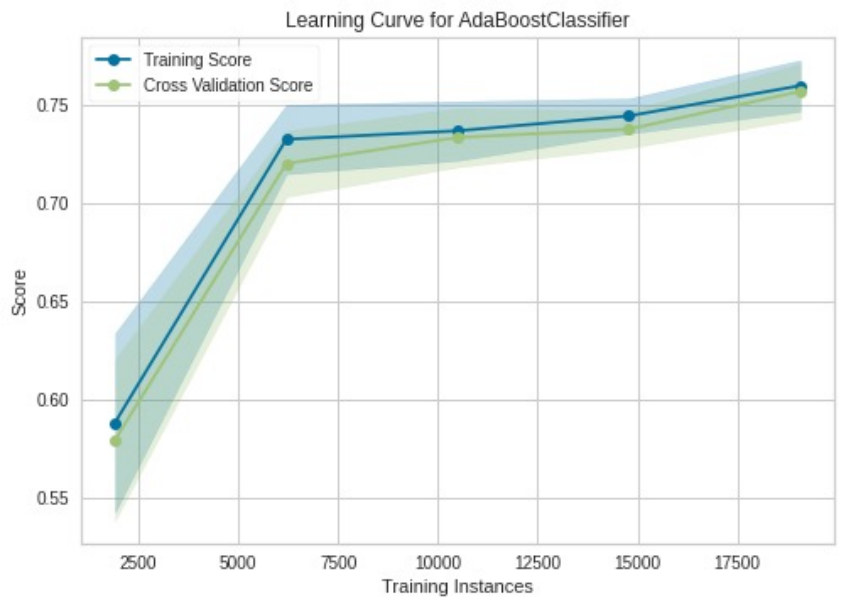
Accuracy	RF	DT	CatBoost	AdBoost	XGBoost
Imbalanced	0.80	0.82	0.87	0.86	0.86
Overbalanced	0.79	0.83	0.78	0.78	0.77
Underbalanced	0.76	0.79	0.77	0.78	0.76
Class weight	0.80	0.81	0.78	N/A	0.86
Threshold	0.80	0.82	0.78	0.86	0.86

Generally higher for imbalanced data and when moving the threshold.

The maximum score derived from CatBoost on original classes.

Multiclass Classification

Overview of the Classes



Trained on 80% of the data; used 20% for testing

Performed 3 fold GridSearchCV on all models except CATBOOST and XGBOOST which have their built in function

Results After Multiclass classification

Score / Model	Random-Forest	AdaBoost	Bagging Classifier	Decision Tree	CatBoost	XGBoost
F1 - train	0.88	0.79	0.63	0.97	0.85	0.84
Accuracy - train	0.88	0.80	0.63	0.97	0.84	0.83
F1 - test	0.83	0.80	0.63	0.77	0.85	0.85
Accuracy - test	0.83	0.79	0.62	0.77	0.87	0.84

Results After Multiclass classification

overfitting

HUGE overfitting

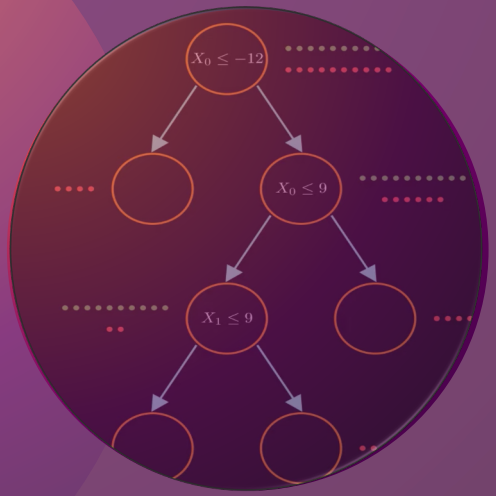
Score / Model	Random-Forest	AdaBoost	Bagging Classifier	Decision Tree	CatBoost	XGBoost
F1 - train	0.88	0.79	0.63	0.97	0.85	0.84
Accuracy - train	0.88	0.80	0.63	0.97	0.84	0.83
F1 - test	0.83	0.80	0.63	0.77	0.85	0.85
Accuracy - test	0.83	0.79	0.62	0.77	0.87	0.84

Results After Multiclass classification

Best models

Score / Model	Random-Forest	AdaBoost	Bagging Classifier	Decision Tree	CatBoost	XGBoost
F1 - train	0.88	0.79	0.63	0.97	0.85	0.84
Accuracy - train	0.88	0.80	0.63	0.97	0.84	0.83
F1 - test	0.83	0.80	0.63	0.77	0.85	0.85
Accuracy - test	0.83	0.79	0.62	0.77	0.87	0.84

Conclusions?



02

Implementation of a Decision Tree From Scratch

Table of contents

Introduction

- We implemented a tree for **Classification** problems and one for **Regression** problems.
- We tested the algorithms on different datasets:
    - For classification: **Iris dataset**
    - For regression: **A self-generated parabola dataset**, with x-values (features), and y-values (labels) and **Diabetes dataset**

Structure of the code

1) Initialization: the Node class

```
class Node():

    def __init__(self, feature_index=None, threshold=None, left=None, right=None, info_gain=None, value=None):

        # A tree has two types of nodes essentially: decision nodes and leaf nodes.

        # Decision nodes:
        self.feature_index = feature_index
        self.threshold = threshold # threshold value for the specific feature

        # right and left are for accessing the left child and right child, in order to move from parent to child
        self.left = left
        self.right = right
        self.info_gain = info_gain # stores the information gain of the split

        # Leaf node
        self.value = value # value is the majority class of the leaf node - will help us predict the class of a new da
```

- **Feature_index**: sepal length (0), sepal width (1), petal length (2), and petal width (3).
- **Threshold**: sets the threshold value for the specific features e.g. ''is X_0 less than or equal to 1.9 threshold value?''
- **Left & Right**: allow to point to the right or left node from the parent
- **Info_gain:** will store the information gain of the split
- Value: comes from the majority class of the leaf node

```
    def make_prediction(self, x, tree):
        ''' function to predict a single data point:

        takes a node as parameter -> we are passing the root node first

        if node.value is not none, meaning if the node is a leaf node, then it returns the value
        if it is not a leaf node, then it will extract the feature value of our new
        data point and the given feature index
        '''

        if tree.value!=None: return tree.value


        feature_val = x[tree.feature_index]

        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)
```

2) Inside DecisionTreeClassifier class

The helper functions to build the trees

```
def split(self, dataset, feature_index, threshold):
    '''
    function to split the data
    '''

    # left side: i pass those feature values smaller than the treshold
    dataset_left = np.array([row for row in dataset if row[feature_index]<=threshold])
    dataset_right = np.array([row for row in dataset if row[feature_index]>threshold])
    return dataset_left, dataset_right
```

```
def information_gain(self, parent, l_child, r_child, mode="entropy"):

    ''' function to compute information gain
        this function basically substracts the comnbined information
        of the child node from the parent node.

        weights: relative sizes of the child with respect to the parent.
        '''

    weight_l = len(l_child) / len(parent)
    weight_r = len(r_child) / len(parent)
    if mode=="gini":
        gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child) + weight_r*self.gini_index(r_child))
    else:
        gain = self.entropy(parent) - (weight_l*self.entropy(l_child) + weight_r*self.entropy(r_child))
    return gain
```

- ***Split***: divides the data into the left and right child nodes based on a threshold value.
- ***Information_gain***: computes the information gain. *«mode»* is set to *entropy* but can be changed to *gini.* In both cases, weight_l and weight_r depend on the **relative sizes of the child with respect to the parent** and the information gain is computed by subtracting the combined and weighted gini indices of the child node from the parent node’s

2) The helper functions to build the trees

```python
def entropy(self, y):

    ''' function to compute entropy '''

    class_labels = np.unique(y)
    entropy = 0
    for cls in class_labels:
        p_cls = len(y[y == cls]) / len(y)
        entropy += -p_cls * np.log2(p_cls)
    return entropy
```

```python
def gini_index(self, y):

    ''' function to compute gini index '''

    class_labels = np.unique(y)
    gini = 0
    for cls in class_labels:
        p_cls = len(y[y == cls]) / len(y)
        gini += p_cls**2
    return 1 - gini
```

- ***Variance_reduction****:* used for the regression tree, instead of computing the information. Defined as the total reduction of the variance of the target variable Y due to the split at that node.

- *entropy* and *gini_index* compute entropy and gini index respectively.

```python
def variance_reduction(self, parent, l_child, r_child):
    ''' function to compute variance reduction '''

    weight_l = len(l_child) / len(parent)
    weight_r = len(r_child) / len(parent)
    reduction = np.var(parent) - (weight_l * np.var(l_child) + weight_r * np.var(r_child))
    return reduction
```

2) The helper functions to build the trees

```python
def get_best_split(self, dataset, num_samples, num_features):
    ''' function to find the best split
    returns a dictionary, indeed best_split dictionary is defined at the beginning and max info gain is initializeed
    as negatve infinity, since we want to maximise the information gain, and to find that we need to use a  number
    bigger that any other number '''

    # dictionary to store the best split
    best_split = {}
    max_info_gain = -float("inf")


    '''
    Here, we will loop over all teh features, through all the possible treshold values.
    N.b. it would not make sense to iterate over every possible real number.
    Indeed, we will iterate through every possible value of a fetaure
    that we have encountered on our dataet: np.unique
    '''

    for feature_index in range(num_features):
        feature_values = dataset[:, feature_index]
        possible_thresholds = np.unique(feature_values)

        # loop over all the feature values present in the data
        for threshold in possible_thresholds:
            # we split the dataset based on the current feature index and the current treshold
            # get current split
            dataset_left, dataset_right = self.split(dataset, feature_index, threshold)
            # check if child not null - we have to make sure they are not empty

            #then we extract the target values (Y)
            if len(dataset_left)>0 and len(dataset_right)>0:
                y, left_y, right_y = dataset[:, -1], dataset_left[:, -1], dataset_right[:, -1]
                # compute information gain
                curr_info_gain = self.information_gain(y, left_y, right_y, "gini")
                # update the best split if needed
                if curr_info_gain>max_info_gain:
                    best_split["feature_index"] = feature_index
                    best_split["threshold"] = threshold
                    best_split["dataset_left"] = dataset_left
                    best_split["dataset_right"] = dataset_right
                    best_split["info_gain"] = curr_info_gain
                    max_info_gain = curr_info_gain

    # return best split
    return best_split
```

- ***Get_best_split**:* finds the point where the data should split. The value of best split gets updated each time.

- ***Calculate_leaf_value**:* calculates the value at the leaf node. It describes the end result or the final predicted value. For classification it follows the majority rule; for regression it return the average value of the predictor column.

```python
def calculate_leaf_value(self, Y):


    ''' function to compute leaf node.

    For regression, leaf node is computed by the mean of the values in the predictor column, while
    for classificaiton, the leaf node is computed by majority rule, i.e. the majority class present
    in a particular node.

    '''

    Y = list(Y)
    return max(Y, key=Y.count)
```

2) The helper functions to build the trees

```
def fit(self, X, Y):

    ''' function to train the tree '''
    # concatenate x and y to create the dataset
    dataset = np.concatenate((X, Y), axis=1)

    # call the build tree function
    #root node will be returned by the build tree fct and it will be stored in self.root
    self.root = self.build_tree(dataset)

def predict(self, X):

    ''' function to predict new dataset '''
    ''' will take a new dataset (matrix of feature) and return the corresponding predictions '''
    preditions = [self.make_prediction(x, self.root) for x in X]

    return preditions

def make_prediction(self, x, tree):
    ''' function to predict a single data point:

    takes a node as parameter -> we are passing the root node first

    if node.value is not none, meaning if the node is a leaf node, then it returns the value
    if it is not a leaf node, then it will extract the feature value of our new
    data point and the given feature index
    '''

    if tree.value!=None: return tree.value


    feature_val = x[tree.feature_index]

    if feature_val<=tree.threshold:
        return self.make_prediction(x, tree.left)
    else:
        return self.make_prediction(x, tree.right)
```

- ***fit*** :concatenates x and y to create the dataset and calls ***build_tree*** to fit it on this latter
- ***Predict***: returns the corresponding predictions calling ***make_predictions***.
- ***Make_prediction:*** takes a node as a parameter and then if node.value is not None, then it will extract the feature value of our new data point and the given feature index.

2) Building the tree

```python
# most important function: it builds the binary tree using recursion
def build_tree(self, dataset, curr_depth=0):

    ''' building the tree '''

    # splits the features and targets into two separate variables
    X, Y = dataset[:,:-1], dataset[:,-1]

    # extracts the number of samples and features
    num_samples, num_features = np.shape(X)

    # split until stopping conditions are met (checks whether n. of samples < min n. of samples)
    if num_samples>=self.min_samples_split and curr_depth<self.max_depth:
        # if these two conditions are not met it will not split further

        # find the best split
        best_split = self.get_best_split(dataset, num_samples, num_features)

        ''' check if information gain is positive
        n.b. info gain = 0 means we are splitting a node which is already pure, i.e. the node consists of only
        one type of class, and that does not make any sesnse
        '''

        if best_split["info_gain"]>0:

            # recursive part (left): we call build_tree

            '''
            will first create left subtree and then once it has reached the leaf node,
            it will create right subtree
            '''

            left_subtree = self.build_tree(best_split["dataset_left"], curr_depth+1)

            # recur right
            right_subtree = self.build_tree(best_split["dataset_right"], curr_depth+1)

            # return decision node
            return Node(best_split["feature_index"], best_split["threshold"],   # best_split is a dictionary
                        left_subtree, right_subtree, best_split["info_gain"])

    # compute leaf node
    leaf_value = self.calculate_leaf_value(Y)
    # return leaf node
    return Node(value=leaf_value)
```

- We first split the independent and dependent variables.
- We also store the *number of samples* and *number of features* of the dataset in two variables.
- The ***number of samples* is a deciding parameter** for us since the data continues to split **until the splitting conditions are met**. Through this iterative process, **it finds the best split** using the *get_best_split* function. It also takes into account if the node is pure or not
- At each decision node we return *threshold*, *information_gain*, the *feature_index* and the *leaf_value* on which the split is decided.

2) Evaluation and Comparison with sklearn

Decision Tree From Sklearn	Decision Tree From Scratch
0.933	0.947

Table 5: Accuracy For Classification Data Set

Decision Tree From Sklearn	Decision Tree From Scratch
0.012	0.094

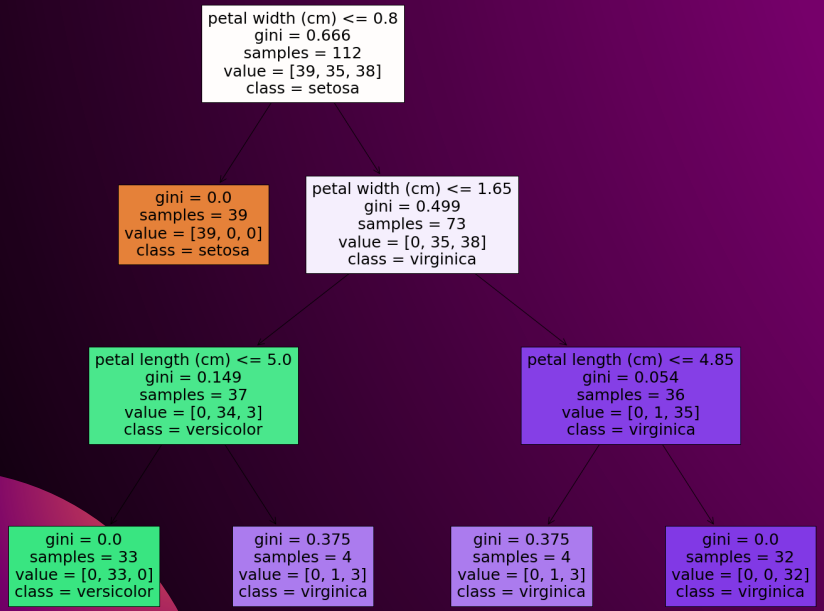
Table 6: MSE For Regression Data Set - Parabola

Decision Tree From Sklearn	Decision Tree From Scratch
4213.98	4506.96

Table 7: MSE For Regression Data Set - Diabetes

- We decided to compare the performance of our tree with that generated from **sklearn**.
- To compare the performance, we used **accuracy** as a measure in case of **classification** and **MSE** for **regression** since it tells us how close our predicted value is to the real point.

Comparison with sklearn-classification

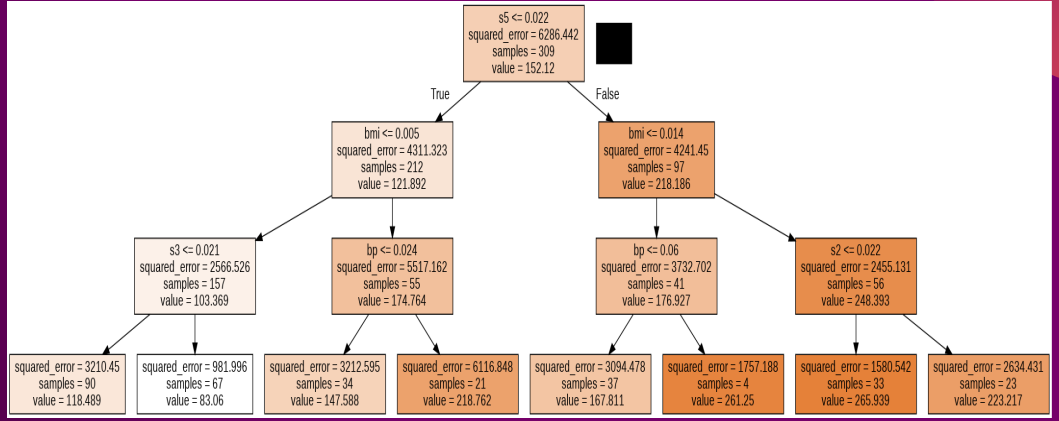



```
X_2 <= 1.9 ? 0.33741385372714494
 left:setosa
 right:X_3 <= 1.5 ? 0.427106638180289
  left:X_2 <= 4.9 ? 0.05124653739612173
    left:versicolor
    right:virginica
  right:X_2 <= 5.0 ? 0.019631171921475288
    left:virginica
    right:virginica
```

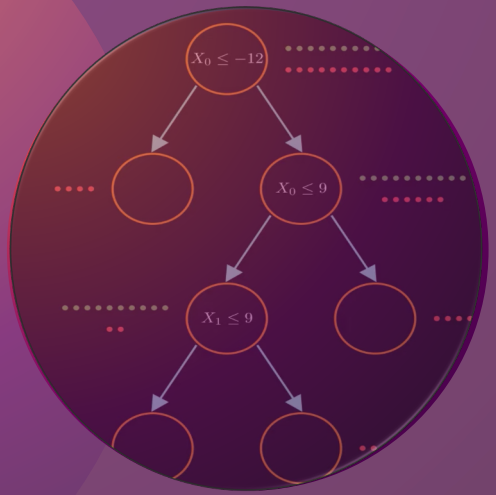
Accuracy for our tree	Accuracy using sklearn
93.3%	94.5%

Comparison with sklearn-regression

```
X_8 <= 0.0213108465682448 ? 1997.0530908365454
 left:X_2 <= 0.00457216660300077 ? 979.3019065841804
  left:X_6 <= 0.0191869970174533 ? 307.072122073178
    left:X_8 <= -0.0439854025655911 ? 377.94398428731756
        left:86.25
        right:130.21212121212122
    right:X_6 <= 0.159089233572762 ? 114.52447734192015
        left:81.74242424242425
        right:170.0
  right:X_3 <= 0.0218723549949558 ? 1195.670590087273
    left:X_6 <= -0.0029028298070691 ? 538.5354409143329
        left:164.72727272727272
        right:116.16666666666667
    right:X_8 <= -0.0332487872476258 ? 1883.3944384771457
        left:85.0
        right:232.8421052631579
 right:X_2 <= 0.0121168511201671 ? 1246.314661475798
  left:X_3 <= 0.0563010619323185 ? 768.6915807836399
    left:X_7 <= 0.0712099797536354 ? 627.3026595391461
        left:159.0909090909091
        right:239.75
    right:X_2 <= -0.0503962491649252 ? 1740.0208333333333
        left:189.0
        right:285.3333333333333
  right:X_5 <= 0.0215459602844172 ? 441.742237705359
    left:X_2 <= 0.0692408910358548 ? 480.98605022557604
        left:251.47826086956522
        right:299.2
    right:X_8 <= 0.062575181458056 ? 709.7381102409458
        left:201.85714285714286
        right:256.44444444444446
```



MSE for our tree	MSE using sklearn
4506.96	4213.98



04
How to run the code

Thank you