ID3 Decision Tree Classifier from scratch in Python

Coding the ID3 algorithm to build a Decision Tree Classifier from scratch.

Bernardo Garcia del Rio Dec 13, 2020 · 7 min read

Photo by Fabrice Villard on Unsplash

In my last article, I showed how you could have some fun **growing a Random Forest using Sklearn's DecisionTreeClassifier**. You can check it out below:

Understanding Decision Trees and Random Forests with a hands-on example

bernardogarciadelrio.medium.com

If you are new to Machine Learning (ML) or are in the process of becoming a Data Scientist or ML practitioner like me, you will probably get something out of it. In my case, preparing the material for the article helped me gain a good understanding of how Random Forests and Decision Trees work under the hood without getting into complex code.

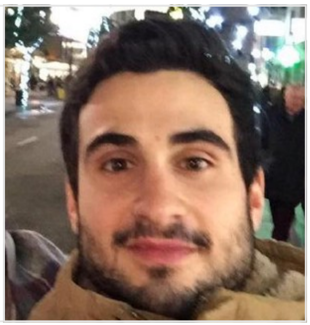
In that article, I mentioned that there are many algorithms that can be used to build a Decision Tree. One of them is **ID3 (Iterative Dichotomiser 3)** and we are going to see how to code it from scratch using ONLY Python to build a Decision Tree Classifier.

All the code can be found in a public repository that I have attached below:

bergr7/ID3_From_Scratch

All the code is in a public repository at the link below: Installation Project Motivation File Descriptions Results...

github.com



Modelling the nodes of the tree

A Decision Tree is formed by nodes: root node, internal nodes and leaf nodes. We can create a Python class that will contain all the information of all the nodes of the Decision Tree.

```python
class Node:
    """Contains the information of the node and another nodes of the Decision Tree."""

    def __init__(self):
        self.value = None
        self.next = None
        self.childs = None

```

nodes.py hosted with ❤ by GitHub view raw

- *value*: Feature to make the split and branches.
- *next*: Next node
- *childs*: Branches coming off the decision nodes

Decision Tree Classifier Class

We create now our main class called DecisionTreeClassifier and use the __init__ constructor to initialise the attributes of the class and some important variables that are going to be needed.

Note that I have provided many annotations in the code snippets that help understand the code.

```
class DecisionTreeClassifier:
    """Decision Tree Classifier using ID3 algorithm."""

    def __init__(self, X, feature_names, labels):
        self.X = X  # features or predictors
        self.feature_names = feature_names  # name of the features
        self.labels = labels  # categories
        self.labelCategories = list(set(labels))  # unique categories
        # number of instances of each category
        self.labelCategoriesCount = [list(labels).count(x) for x in self.labelCategories
        self.node = None  # nodes
        # calculate the initial entropy of the system
        self.entropy = self._get_entropy([x for x in range(len(self.labels))])

```

decisiontree.py hosted with ❤ by GitHub view raw

In order to calculate the entropy note we are using the private function _get_entropy( ). The code for this function is provided below:

```
def _get_entropy(self, x_ids):
    """ Calculates the entropy.
    Parameters
    __________
    :param x_ids: list, List containing the instances ID's
    __________
    :return: entropy: float, Entropy.
    """
    # sorted labels by instance id
    labels = [self.labels[i] for i in x_ids]
```

```python
    entropy = sum([-count / len(x_ids) * math.log(count / len(x_ids), 2)
                    if count else 0
                    for count in label_count
                   ])

        return entropy

```

entropy.py hosted with ❤ by GitHub view raw

We pass the instances id's or indexes to this function. For doing this, we need to generate an unique number for each instance. Python's lists comprehensions come in very handy for this task as you can see.

We are going to code an ID3 algorithm that uses the **information gain** to find the feature that maximises it and make a split based on that feature. The information gain is based on entropy.

As this article is about coding the ID3 algorithm, I am not going to go into the details but **entropy** is a measure of uncertainty about a random variable. If we minimise the entropy, then we increase the certainty about the variable. In other words, if the random variable can take only one value the entropy reaches its minimum whereas if all the values are equiprobable the entropy is maximum.

> *The algorithm will try to minimise the entropy or, equivalently, maximising the information gain.*

We can compute the entropy with the following formula:

$$H(S) = \sum_{i=1}^{N} -p_i \log_2 p_i$$

Entropy formula (Caption by Author)

where *pi* is the proportion of each category i=1,…,N.

We said that we would compute the information gain to choose the feature that maximises it and then make the split based on that feature. The information gain is a

$$G(S,j) = H(S) - \sum_j \frac{|S_j|}{|S|} H(S_j)$$

Information gain (Caption by Author)

So we create another private function that computes the information gain:

```
def _get_information_gain(self, x_ids, feature_id):
    """Calculates the information gain for a given feature based on its entropy and the
    Parameters
    __________
    :param x_ids: list, List containing the instances ID's
    :param feature_id: int, feature ID
    __________
    :return: info_gain: float, the information gain for a given feature.
    """
    # calculate total entropy
    info_gain = self._get_entropy(x_ids)
    # store in a list all the values of the chosen feature
    x_features = [self.X[x][feature_id] for x in x_ids]
    # get unique values
    feature_vals = list(set(x_features))
    # get frequency of each value
    feature_v_count = [x_features.count(x) for x in feature_vals]
    # get the feature values ids
    feature_v_id = [
        [x_ids[i]
        for i, x in enumerate(x_features)
        if x == y]
        for y in feature_vals
    ]

    # compute the information gain with the chosen feature
    info_gain_feature = sum([v_counts / len(x_ids) * self._get_entropy(v_ids)
                        for v_counts, v_ids in zip(feature_v_count, feature_v_id)])

    info_gain = info_gain - info_gain_feature

    return info_gain

```

infogain.py hosted with ❤ by GitHub view raw the feature specified in feature_id and finally the information gain.

This ID3 algorithm chooses the feature that maximise the information gain at each split. That is if the amount information in the feature is large, the entropy will be small, and therefore the second term of the information gain formula will be also small, increasing the information gain.

_get_information_gain( ) only calculates the information gain for a given set and feature but, at each split, we need to figure out which is the **feature** that maximises the information gain. We need to create another function to find the feature that maximises the information gain.

```
def _get_feature_max_information_gain(self, x_ids, feature_ids):
    """Finds the attribute/feature that maximizes the information gain.
    Parameters
    __________
    :param x_ids: list, List containing the samples ID's
    :param feature_ids: list, List containing the feature ID's
    __________
    :returns: string and int, feature and feature id of the feature that maximizes the i
    """
    # get the entropy for each feature
    features_entropy = [self._get_information_gain(x_ids, feature_id) for feature_id in
    # find the feature that maximises the information gain
    max_id = feature_ids[features_entropy.index(max(features_entropy))]

    return self.feature_names[max_id], max_id
```

getmaxinfogain.py hosted with ❤ by GitHub view raw

Ok..so we have a private function (**_get_feature_max_information_gain( )**) that find the feature that maximises the information gain, that uses another private function (**_get_information_gain( )**) that calculates the information gain, that uses another private function (**_get_entropy()**) that computes the entropy. We are all set! We can start coding the ID3 algorithm that will create our ID3 Decision Tree for classification problems.

We create a function that initialises the algorithm and then uses a private function to call the algorithm recursively to build our tree.

```
def id3(self):
    """Initializes ID3 algorithm to build a Decision Tree Classifier.
```

```python
        # assign an unique number to each instance
        x_ids = [x for x in range(len(self.X))]
        # assign an unique number to each featuer
        feature_ids = [x for x in range(len(self.feature_names))]
        # define node variable - instance of the class Node
        self.node = self._id3_recv(x_ids, feature_ids, self.node)
```

id3.py hosted with ❤ by GitHub view raw

Note how we call **_id3_recv( )** function in *self.node*. This function returns an instance of the class Node with information of all the nodes (decisions) in the Decision Tree.

_id3_recv_( ) is the trickiest function to code so let's spend some time understanding what is does.

Let's generate some data

I have generated a small toy dataset so we can work on a small example and actually visualise the resulting tree. This will help us understand the algorithm as well.

Let's imagine that we love surfing waves and would like to have a model to predict if there will be good waves in our favourite spot.

Our features are:

Feature	Values
wind_direction	{'N', 'S', 'E', 'W'}
tide	{'high', 'low'}
swell_forecasting	{'small', 'medium', 'large}

Features (Caption by Author)

And our target is simply a binary categorial variable that tells us if there were/will be good waves based on the information provided in our features:

Target	Values
good_waves	{'Yes', 'No'}

Target (Caption by Author)

Let's generate some synthetic data using random sampling:

```
    'tide': ['Low', 'High'],
    'swell_forecasting': ['small', 'medium', 'large'],
    'good_waves': ['Yes', 'No']
}

# create an empty dataframe
data_df = pd.DataFrame(columns=data.keys())

np.random.seed(42)
# randomnly create 1000 instances
for i in range(1000):
    data_df.loc[i, 'wind_direction'] = str(np.random.choice(data['wind_direction'], 1)[0
    data_df.loc[i, 'tide'] = str(np.random.choice(data['tide'], 1)[0])
    data_df.loc[i, 'swell_forecasting'] = str(np.random.choice(data['swell_forecasting']
    data_df.loc[i, 'good_waves'] = str(np.random.choice(data['good_waves'], 1)[0])

data_df.head()
```

data.py hosted with ❤ by GitHub view raw

	wind_direction	tide	swell_forecasting	good_waves
0	E	Low	large	No
1	E	Low	large	No
2	W	Low	small	No
3	N	High	small	No
4	N	High	small	Yes

First 5 rows of our toy dataset (Caption by Author)

_id3_recv( ) function

Now that we have some data to work with it will be easier to understand _id3_recv ( ) function. Below is the code with some annotations:

```
def _id3_recv(self, x_ids, feature_ids, node):
    """ID3 algorithm. It is called recursively until some criteria is met.
    Parameters
    __________
    :param x_ids: list, list containing the samples ID's
    :param feature_ids: list, List containing the feature ID's
    :param node: object, An instance of the class Nodes
    __________
    :returns: An instance of the class Node containing all the information of the nodes
    """
    if not node:
```

```python
        # if all the example have the same class (pure node), return node
        if len(set(labels_in_features)) == 1:
            node.value = self.labels[x_ids[0]]
            return node
        # if there are not more feature to compute, return node with the most probable class
        if len(feature_ids) == 0:
            node.value = max(set(labels_in_features), key=labels_in_features.count)  # compu
            return node
        # else...
        # choose the feature that maximizes the information gain
        best_feature_name, best_feature_id = self._get_feature_max_information_gain(x_ids, f
        node.value = best_feature_name
        node.childs = []
        # value of the chosen feature for each instance
        feature_values = list(set([self.X[x][best_feature_id] for x in x_ids]))
        # loop through all the values
        for value in feature_values:
            child = Node()
            child.value = value  # add a branch from the node to each feature value in our f
            node.childs.append(child)  # append new child node to current node
            child_x_ids = [x for x in x_ids if self.X[x][best_feature_id] == value]
            if not child_x_ids:
                child.next = max(set(labels_in_features), key=labels_in_features.count)
                print('')
            else:
                if feature_ids and best_feature_id in feature_ids:
                    to_remove = feature_ids.index(best_feature_id)
                    feature_ids.pop(to_remove)
                # recursively call the algorithm
                child.next = self._id3_recv(child_x_ids, feature_ids, child.next)
        return node
```

id3recv.py hosted with ❤ by GitHub view raw

The function takes two lists containing instances ids and feature ids and an instance of the class Node. Firstly, it checks if the node attribute is *None* and if it is, it initialises an instance of the class Node.

Then, at each split the algorithm checks if some of the following conditions are met:

- **Are all the instances of the same category?** If so, it assigns the category to the node value and returns the variable node. The following chunk of code does this task in the function:

```python
# if all the example have the same class (pure node), return node
```

```
            return node
```

condition1.py hosted with ❤ by GitHub view raw

- **Are there more features to compute the information gain?** If there are not, it assigns the most probably category to the node value (it computes the mode) and returns the variable node. Code doing this is below:

```
# if there are not more feature to compute, return node with the most probable class
        if len(feature_ids) == 0:
            node.value = max(set(labels_in_features), key=labels_in_features.count)  # cc
            return node
```

condition2.py hosted with ❤ by GitHub view raw

If none of the above conditions are met, it chooses the feature that maximises the information gain and stores it in the node value. In this case, the node value is the **feature** with which it is making the split.

The ID3 algorithm creates a branch for each value of the selected feature and finds the instances in the training set that takes that branch. Note each branch is represented with a new instance of the class node that also contains the the next node.

For the next node, it computes the information gain for the remaining features and instances and chooses the one that maximises it to make another split until all the instances have the same class or there not more features to compute, returning the most probable category in this case. All of this is done in the last part of the function:

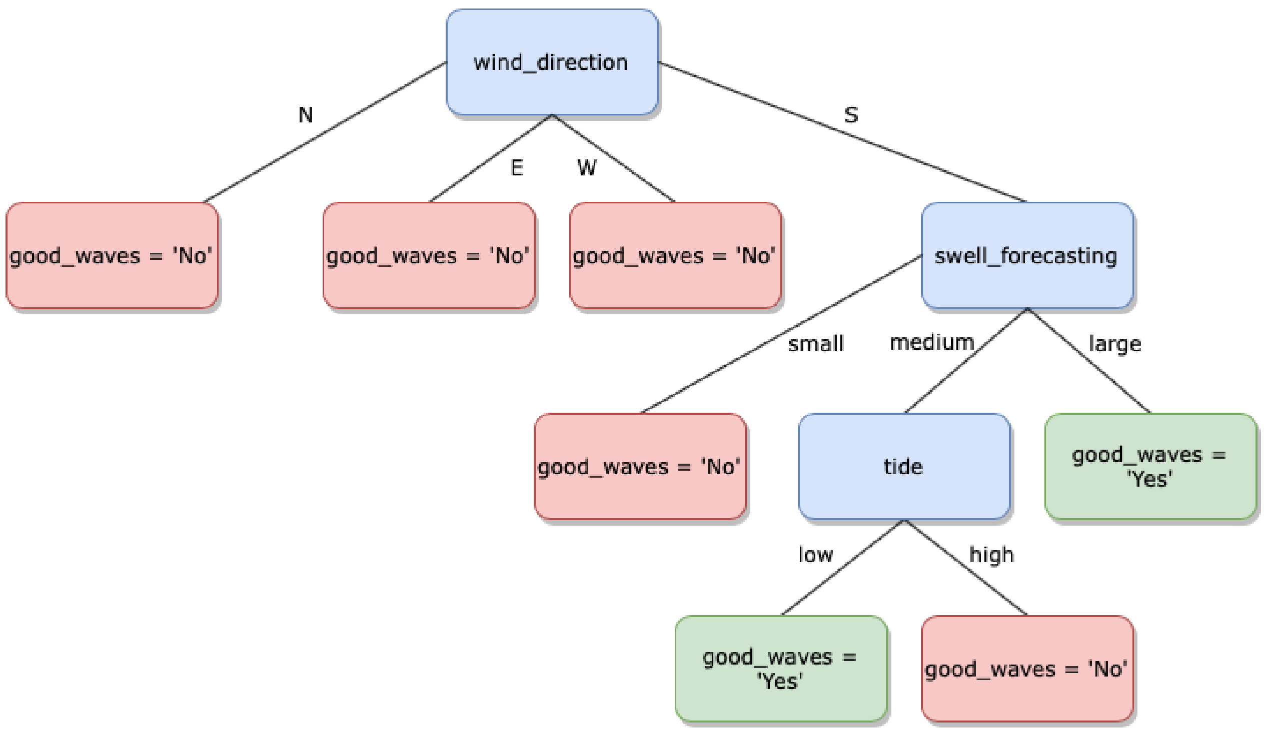
```
        # else...
        # choose the feature that maximizes the information gain
        best_feature_name, best_feature_id = self._get_feature_max_information_gain(x_ids,
        node.value = best_feature_name
        node.childs = []
        # value of the chosen feature for each instance
        feature_values = list(set([self.X[x][best_feature_id] for x in x_ids]))
        # loop through all the values
        for value in feature_values:
            child = Node()
            child.value = value  # add a branch from the node to each feature value in our
            node.childs.append(child)  # append new child node to current node
            child_x_ids = [x for x in x_ids if self.X[x][best_feature_id] == value] # insta
            if not child_x_ids:
                child.next = max(set(labels_in_features), key=labels_in_features.count)
```

```
                    to_remove = feature_ids.index(best_feature_id)
                    feature_ids.pop(to_remove)
                # recursively call the algorithm
                child.next = self._id3_recv(child_x_ids, feature_ids, child.next)
        return node
```

id3process.py hosted with ❤ by GitHub view raw

Resulting Decision Tree

And here we have the Decision Tree Classifier that the ID3 algorithm has built using our synthetic data:



Resulting Decision Tree (Caption by Author)

It seems that our “synthetic” beach only works well when the wind blows from the South. If there is a large swell, it is likely that there will be good waves regardless of the tide but if the swell is medium, we will usually find better surfing conditions during low tide.

I would like to give credit to the Professors of the MSc in Machine Learning, Deep Learning and AI by Big Data International Campus for providing a version of the source code, that inspired me to write this article.

International Campus and UCAM

Decision Tree | Decision Tree Classifier | Machine Learning | Id3 Algorithm | Artificial Intelligence