Aldo Hernández

DECISION TREES WITH PYTHON

Abstract *This document explores decision tree algorithms for classification tasks, focusing on Boolean classification where inputs have discrete values and outputs have two possible values. The document explains how decision trees function through sequential tests at each node, with branches representing attribute values and leaf nodes providing final decisions. It discusses the challenges of representing all Boolean functions efficiently and the exponential growth of possible functions as attributes increase. The methodology section details a practical implementation using a dataset of songs, predicting whether they reached the top position on Billboard charts. The process includes data preprocessing to handle missing values, mapping categorical variables to numerical ones, and determining optimal tree depth through k-fold cross-validation. The resulting decision tree achieves 71.18% accuracy and successfully predicts that Camila Cabello's "Havana" would reach number one. The paper concludes that decision trees are powerful classification models that perform best with large, complete datasets, though they require careful handling of categorical and continuous variables. Through thorough preliminary analysis, the implemented model demonstrates reliable predictive capability for music chart performance.*

Keywords decision tree, machine learning, python, cross-validation

1. Introduction

A decision tree is a function that takes as input a vector of attribute values and returns a single output value—also called decision—. The input and output values can be discrete or continuous [2]. When inputs have discrete values and the output has exactly two possible values it is called a **Boolean classification**, since any example input will be classified as true or false.

For the decision tree to reach a decision, it has to perform a sequence of tests. Each tree node corresponds to a test of the value of one of the input attributes (A_i), and the branches from the node are labeled with the possible values for said attributes ($A_i = v_{ik}$). Each *leaf node* specifies a value to be returned by the function [2].

This representation is very common for humans; for example, when there is a high rain probability and the question "should I take an umbrella?" arises, how we decide to take it or not is essentially a decision tree. These decisions are of the type "if this, then that" [1].

A Boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the paths leading to a leaf with value *true* [2]

$$Goal \Leftrightarrow (Path_1 \vee Path_2 \vee \dots)$$

$$Path_i = (A_i = v_{ik} \wedge A_j = v_{jk} \wedge \dots)$$

For a wide variety of problems, the decision tree format has a nice concise result. But some functions cannot be represented concisely. So decision trees are good for some kinds of functions and bad for others. Now, consider the set of all Boolean functions on n attributes, how many different functions are in this set? The answer is basically how many different true tables can be written down since the function is defined by its truth table. A truth table over n attributes has 2^n rows (one for each combination of attributes). The "answer" column of the truth table can be considered as a 2^n-bit number that defines the function, which means that there are 2^{2^n} different functions since each function has a version that outputs *true* for that specific 2^n-bit number and another version that outputs *false* for the same number. The reader can see that this number gets exponentially big as we have more and more attributes, and we will have an even larger number of possible trees to choose from since multiple trees can compute the same function. So it is impossible to find *any* kind of representation that is efficient for *all* kinds of functions. [2]

For this algorithm, it may seem obvious that we want a tree that is consistent with the examples and is as small as possible. But because of what was stated above, there is no efficient way to search through the more than 2^{2^n} trees for the tree that we want. However, with some heuristics it is possible to find a good approximate solution: a small consistent tree. The **decision-tree-learning** algorithm adopts a **greedy divide-and-conquer** strategy: always test the most important—the one that makes the most difference when classifying—attribute first [2]. This test divides the problem up into smaller subproblems that can be then solved recursively.

After the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, there are four cases to consider for these recursive problems:

- If the remaining examples are all positive or all negative, the decision can be done.
- If there are some positive and some negative examples, then choose the best attribute to split them.
- If there are no examples left, no example has been observed for this combination of values, so we return a default value.
- If there are no attributes left, but both positive and negative examples, it means that said examples have the exact same description, but different classifications. This can happen because there is an error or **noise** in the data; because the domain is nondeterministic; or because we couldn't observe an attribute that would distinguish the examples. The best we can do is to return the most frequent type of the remaining examples, i. e. if you have left 3 positives and 2 negatives, return positive.

It is important to note that this algorithm does not look at the *correct* Boolean functions, instead, it looks at the given examples to draw a simple small-fitting tree that is consistent with all the examples.

The accuracy of a learning algorithm can be evaluated with a **learning curve**. For a decision tree in a problem domain, the curve shows that as the training set size grows, accuracy increases [2].

The greedy search used in this type of learning is designed to minimize the depth of the final tree, the idea is to pick the attribute that goes as far as possible towards providing an exact classification of the examples. A perfect attribute will divide the examples into sets that area all positive or all negative—and thus, be leaves of the tree—[2]. Now, the question arises, how can we measure this? For that, we will use the notion of information gain, defined in terms of **entropy**.

Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy [2]. In general, the entropy of a random variable V with values v_k, each with probability $P(v_k)$ is defined as

$$H(V) = \sum_k P(V_k) \log_2 \left(\frac{1}{P(v_k)} \right) = - \sum_k P(v_k) \log_2 P(v_k)$$

It will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability q:

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q))$$

Going back to decision tree learning, if a training set contains p positive examples and n negative examples, then the entropy of the goal attribute on the whole set is

$$H(Goal) = B\left(\frac{p}{p+n}\right)$$

A test on a single attribute A might give us only part of the whole entropy bits of the problem. We can measure exactly how much by looking at the entropy remaining after the attribute test.

An attribute A with d different values divides the training set E into subsets $E_1, \ldots, E_d$. Each subset E_k has p_k positive examples and n_k negative examples, so if we go along the E_d branch, we will need an additional $B(p_k/(p_k + n_k))$ bits of information to answer the question [2]. The probability that a randomly selected example from the training set that has the k-th value for said attribute is $(p_k + n_k)/(p + n)$, so the expected entropy remaining after testing the attribute A is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B\left(\frac{p_k}{p_k + n_k}\right)$$

The **information gain** from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B\left(\frac{p}{p+n}\right) - Remainder(A)$$

When dealing with continuous values, we may have to use and minimize the **GINI impurity** index to decide which attribute use next to split the examples, since it will measure how unordered the nodes will be after splitting them up [1]. However, we will keep focusing only in categorical classification. On some problems, this algorithm will generate a large tree when there is no pattern to be found since it will seize on any ”pattern” it can find in the input. This problem is called **overfitting** and occurs in all type of learners. Overfitting is more likely as the hypothesis space and number of input attributes grows, and less likely as we increase the number of training examples.

For decision trees, there is a technique called **decision tree pruning** that combats overfitting. It works by eliminating nodes that are not clearly relevant. We start with a full tree generated by decision-tree-learning, then we look at a test node that has *only* leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data—then we eliminate the test replacing it with a leaf node [2].

But how can we detect a node is testing an irrelevant attribute? If a node were to be irrelevant, we would expect it to split examples into subsets with roughly the same proportion of positive examples as the whole set $(p/(p + n))$ so that the *information gain* will be close to zero. Then, a new question arises, how large should a gain be?

For this, we could use a significance test by assuming there is no underlying pattern (null hypothesis). Then, the actual data is analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (less than 5% of probability), then it is considered to be good evidence for the presence of a significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling [2].

With pruning, noise in examples can be tolerated [2]. Pruned trees perform significantly better than unpruned trees when the data contains a larga amount of noise. Also, they are often smaller and easier to understand.

Another technique is called **early stopping** and combines pruning with information gain, it consists in having the decision tree algorithm to stop generating nodes when there is no good attribute to split on, instead of generating them all and then pruning them away. The problem with this approach is that it stops us from recognizing situations when there is no good attribute, but there are *combinations* of attributes that are informative.

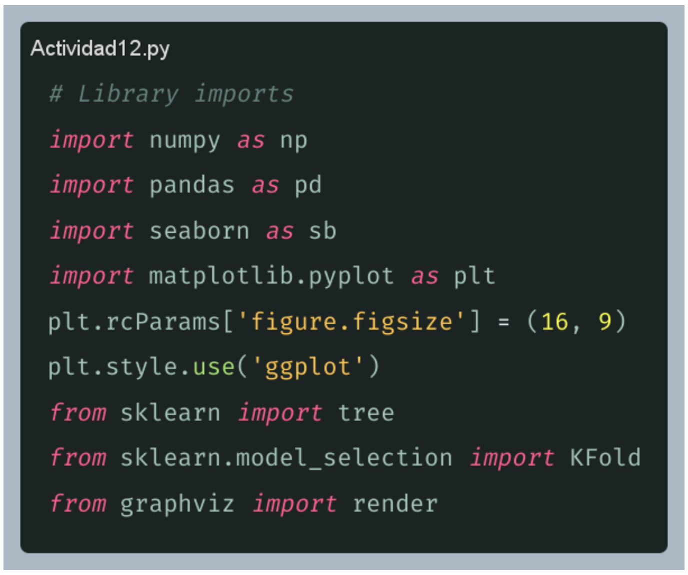
There are several issues with decision trees:

- Missing data: Sometimes, not all attributes values will be known for every example.
- Multivalued attributes: When an attribute has many possible values, the information gain measure gives an inappropriate indication of the attribute's usefulness.
- Continuous and integer-valued input attributes: They have an infinite set of possible values. Rather than generating infinitely many branches, these algorithms typically find the split point that gives the highest information gain.
- Continuous-valued output attributes: When trying to predict a numerical output, we need a regression tree rather than a classification tree. This tree has at each leaf a linear function of some subset of numerical attributes rather than a single value.

2. Methodology

2.1. Before typing code

In order to work with our data, we'll need to download the .csv file [1] to use it as our dataset along with the Graph Visualization Software. Then, we have to make some library imports



Actividad12.py

```python
# Library imports
import numpy as np
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (16, 9)
plt.style.use('ggplot')
from sklearn import tree
from sklearn.model_selection import KFold
from graphviz import render
```

Figure 1. Importing libraries.

2.2. Initial analysis

Now, we process the csv file as a Pandas dataset. After that, we see that our data is very unbalanced since we have a lot of songs that did not make the top 1 on billboard charts in comparison to songs that made the top 1.

Also, when examining the year of birth of artists, we see that we have a lot of artists that have 0 as their year of birth. This is data that we do not have so we need to fix this in the next subsection.

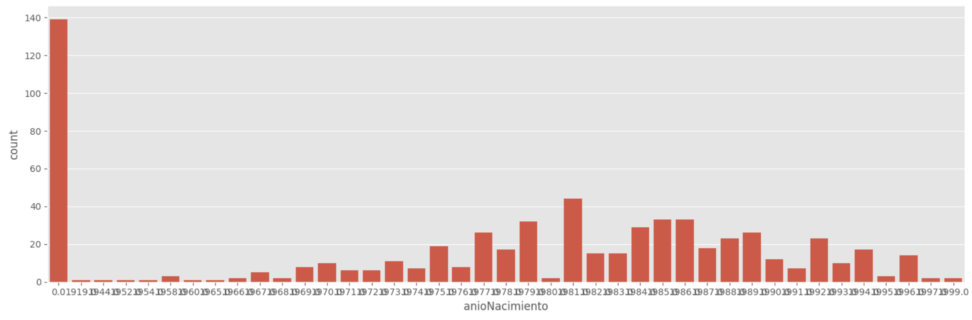
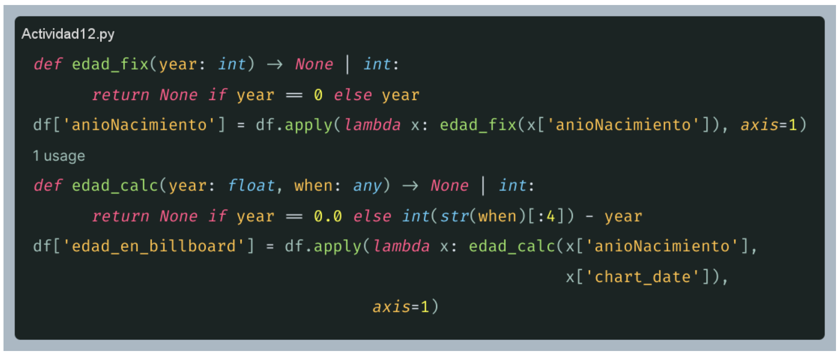


Figure 2. Row count per year of birth

2.3. Preparing the data

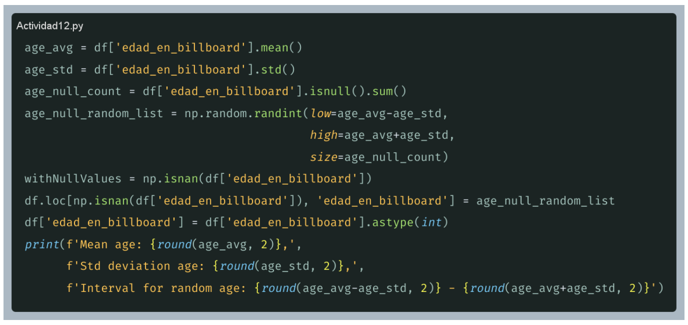
The idea behind having a year of birth column is to know the age of the artist at the time they released their song that entered the billboard chart. First, we will replace all zeroes for None with a function that will be applied to the whole dataset. Then, we will use another function to obtain the artist age when the song was released (or None if we don't know the age of the artist).



```python
def edad_fix(year: int) -> None | int:
        return None if year == 0 else year
df['anioNacimiento'] = df.apply(lambda x: edad_fix(x['anioNacimiento']), axis=1)
def edad_calc(year: float, when: any) -> None | int:
        return None if year == 0.0 else int(str(when)[:4]) - year
df['edad_en_billboard'] = df.apply(lambda x: edad_calc(x['anioNacimiento'],
                                                      x['chart_date']),
                                   axis=1)
```

Figure 3. Fixing the dataset rows.

Next, we will obtain the mean and standard deviation of the ages that we know and generate random numbers within the range of the mean minus deviation to mean plus deviation. This is because statistically the majority of data should be concentrated around these values.



Actividad12.py

```python
age_avg = df['edad_en_billboard'].mean()
age_std = df['edad_en_billboard'].std()
age_null_count = df['edad_en_billboard'].isnull().sum()
age_null_random_list = np.random.randint(low=age_avg-age_std,
                                         high=age_avg+age_std,
                                         size=age_null_count)
withNullValues = np.isnan(df['edad_en_billboard'])
df.loc[np.isnan(df['edad_en_billboard']), 'edad_en_billboard'] = age_null_random_list
df['edad_en_billboard'] = df['edad_en_billboard'].astype(int)
print(f'Mean age: {round(age_avg, 2)},',
      f'Std deviation age: {round(age_std, 2)},',
      f'Interval for random age: {round(age_avg-age_std, 2)} - {round(age_avg+age_std, 2)}')
```

Figure 4. Filling unknown ages.

We can visualize real values and filled ones in the following plot.

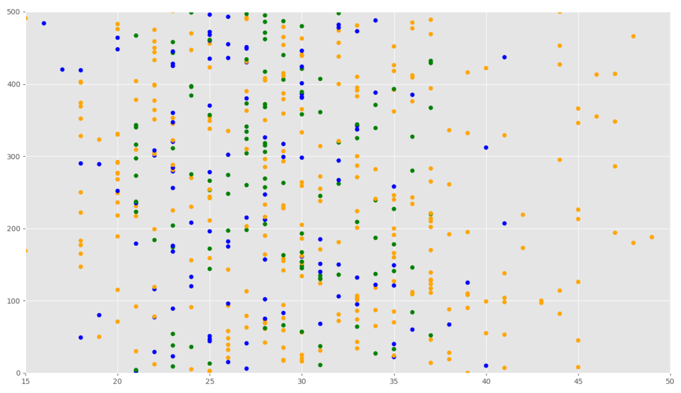
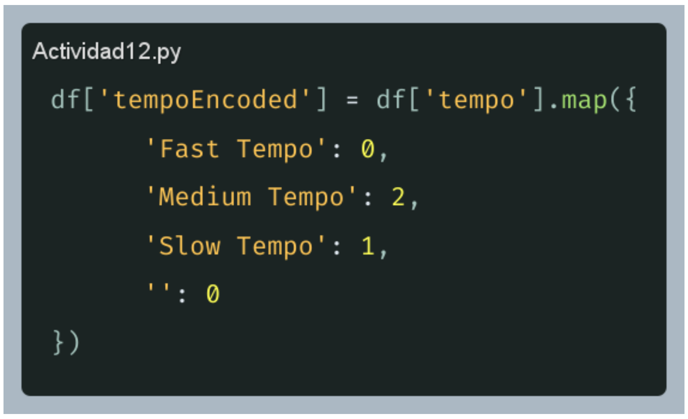


Figure 5. Artist age when they released: top 1 songs (blue), no top 1 songs (yellow), and filled year songs (green).

2.4. Mapping data

Another problem with our data is that we have too many categorical variables—such as mood, tempo, genre, and artist type—and continuous variables—like artist age and duration—. To fix this, we have to **map**—assign a value—each of these categories to a discrete numerical value. Essentially we will then just drop all the old columns since we won't be able to use them for creating our tree.



Actividad12.py

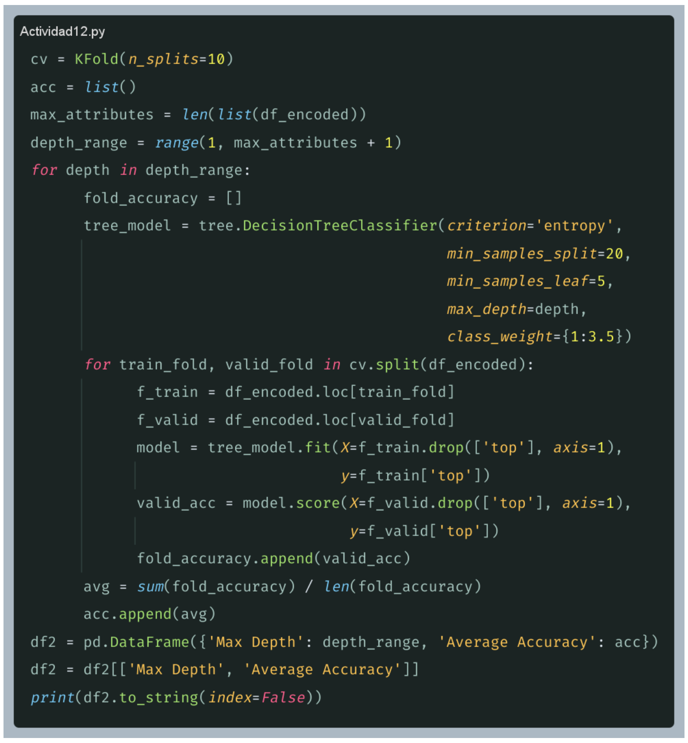
```
df['tempoEncoded'] = df['tempo'].map({
      'Fast Tempo': 0,
      'Medium Tempo': 2,
      'Slow Tempo': 1,
      '': 0
})
```

Figure 6. Mapping the tempo.

2.5. Creating the tree

To test for model accuracy, we need to test the model on a set of examples it has not seen yet. The first approach to do this is to use **holdout cross-validation** that consists of randomly splitting the data into a training set that produces the hypothesis h and a validation set that tests the h. This approach has a clear disadvantage: it does not use all the available data to get the best hypothesis. A good alternative is to use **k-fold cross-validation**, the idea of this method is that each example serves as training and test data. First, the data will be split into k equal subsets to then perform k rounds of learning where in each round $1/k$ of the data is used as test set and the remaining examples will be used as training data [2].

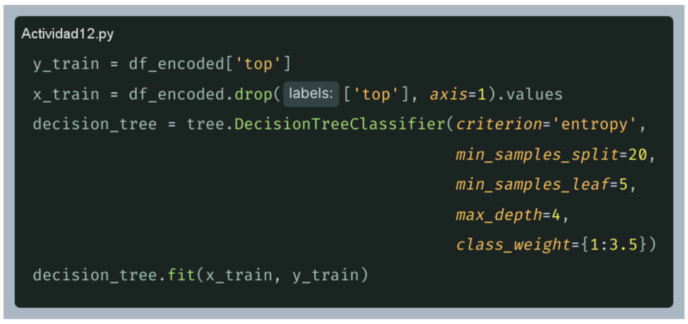
We have to find the most appropriate depth for our tree, for this we will iterate for each possible depth from 1 to 7 (maximum quantity of attributes) creating a tree model with said depth, then using 10-fold cross-validation we will fit and test the model to get an average accuracy.


Actividad12.py

```python
cv = KFold(n_splits=10)
acc = list()
max_attributes = len(list(df_encoded))
depth_range = range(1, max_attributes + 1)
for depth in depth_range:
    fold_accuracy = []
    tree_model = tree.DecisionTreeClassifier(criterion='entropy',
                                             min_samples_split=20,
                                             min_samples_leaf=5,
                                             max_depth=depth,
                                             class_weight={1:3.5})
    for train_fold, valid_fold in cv.split(df_encoded):
        f_train = df_encoded.loc[train_fold]
        f_valid = df_encoded.loc[valid_fold]
        model = tree_model.fit(X=f_train.drop(['top'], axis=1),
                               y=f_train['top'])
        valid_acc = model.score(X=f_valid.drop(['top'], axis=1),
                                y=f_valid['top'])
        fold_accuracy.append(valid_acc)
    avg = sum(fold_accuracy) / len(fold_accuracy)
    acc.append(avg)
df2 = pd.DataFrame({'Max Depth': depth_range, 'Average Accuracy': acc})
df2 = df2[['Max Depth', 'Average Accuracy']]
print(df2.to_string(index=False))
```

Figure 7. Creating models to see the best depth.

After doing this, we can see that the highest accuracy in average can be achieved with a depth of 4. Then, we create the tree model and fit it to all our examples, using a class weight ratio of 1 : 3.5 since the tree is not balanced (it is the ratio of top 1 songs to no top 1 songs).



Actividad12.py

```
y_train = df_encoded['top']
x_train = df_encoded.drop( labels: ['top'], axis=1).values
decision_tree = tree.DecisionTreeClassifier(criterion='entropy',
                                            min_samples_split=20,
                                            min_samples_leaf=5,
                                            max_depth=4,
                                            class_weight={1:3.5})
decision_tree.fit(x_train, y_train)
```

Figure 8. Creating the tree with a depth of 4 and class weight.

We get a decision tree with an accuracy of 71.18%, and we can visualize it in the next figure.

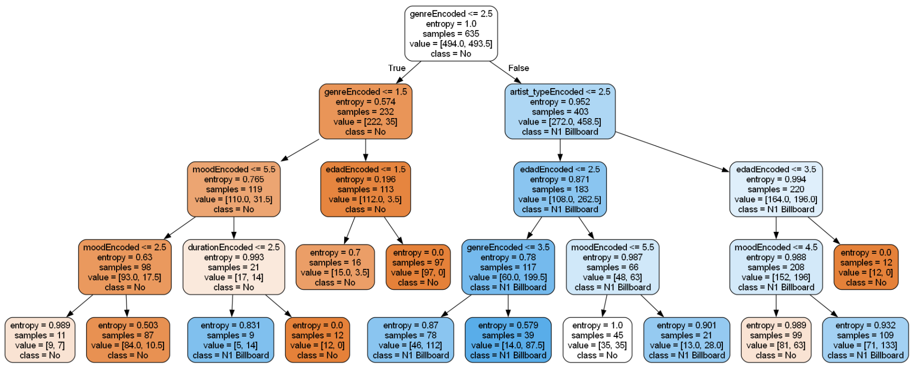


Figure 9. The decision tree.

3. Results

We can see in Figure 9 there are questions that depending on the answer, reduce the entropy by a lot or not. For example, if the genre is not urban or pop (true path for the root node) the entropy is reduced almost by half, which indicated that a lot of top 1 songs are urban or pop.

As for predictions, we can predict if the song Havana by Camila Cabello would make it to the top 1 of the billboard chart. The model says that it will with an 86.21% of accuracy, and the model is correct since this song reached top 1 in 2017.

4. Conclusions

The decision tree is a very powerful model for classification, but can be a little tricky when working with categorical or continuous variables. As with every model, it works best as the sample gets bigger and does not contain missing data.

Since we did a lot of analysis before actually creating the model, the created decision tree became a very trustworthy model for predictions.

References

[1] Bagnato J.I.: *Aprende Machine Learning en Español.* Leanpub, 1.5st ed., 2020.

[2] Russell S., Norvig P.: *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd ed., 2010.

Affiliations

Aldo Hernández
Universidad Autónoma de Nuevo León, San Nicolás de los Garza,
aldo.hernandezt@uanl.edu.mx

Received: ???
Revised: ???
Accepted: ???