Predict the Category of Crimes using Decision Tree

Justin Dula
Computer Science
The University of Texas at Dallas
800 W Campbell Rd, Richardson, TX 75080, United States
justin@utdallas.edu

Adharsh Rajendran
Computer Science
The University of Texas at Dallas
800 W Campbell Rd, Richardson, TX 75080, United States
axr150830@utdallas.edu

Zixuan Yang
Computer Science
The University of Texas at Dallas
800 W Campbell Rd, Richardson, TX 75080, United States
zxy190004@utdallas.edu

Yuncheng Gao
Computer Science
The University of Texas at Dallas
800 W Campbell Rd, Richardson, TX 75080, United States
yxg151230@utdallas.edu

***Abstract*—A distributed decision-tree system is implemented in PySpark on a substantial dataset. The decision tree algorithm and key understandings for its use in the real world are presented, and then applied to a substantial real-world dataset. The results of this experiments are then analyzed and discussed.**

***Keywords*—decision trees, machine learning, Spark, Python**

I. Introduction

Distributed methodologies for machine learning are growing in necessity. One popular machine learning method is decision trees, which we will perform an analysis of from the perspective of the base algorithm, before discussing components which are potential candidates for distribution and parallelization.

We shall then perform a test of the decision tree algorithm on a distributed dataset using Spark and analyze the results, as well as show the tree learned over the data. Finally, we conclude by discussing potential future directions and areas to experiment in with distributed decision trees.

II. Background and Datasets

A. San Fransisco Crime Dataset

Initially, we planned on using the San Francisco crime dataset [2]. However, initial testing with decision trees on the dataset rapidly revealed that it was very poorly suited for decision trees, as it required pattern recognition over small groups of data and had more class labels than attributes.

B. Titanic Dataset

After determining that the San Francisco crime dataset [2] was not suitable for our method, we pivoted to the Titanic dataset [3]. The Titanic dataset is divided into a train and test split, where the train split has 891 rows, and the test split has 417 rows.

Each row corresponds to a single passenger on the Titanic. The data has a binary "Survival" target, where a 1 indicates survival and a 0 indicates that the passenger did not survive. The data has a mix of continuous and discrete attributes.

C. PySpark

We elected to implement the project in PySpark as opposed to Scala to improve iteration speed and take advantage of the learning resources for PySpark. Additionally, several of our team members are more comfortable in Python then Scala, reducing development difficulty.

III. Decision Trees

A decision tree is used to describe a function that transforms an input vector to an output class via a series of decisions [1], each of which recursively repeats this process, making "decisions" until a leaf is reached, at which point the leaf is used to determine the output class. Typically, decision trees are binary trees, where each internal node (including the root) represents a logical Boolean test over some attribute of the dataset. The decision partitions the input into two groups, one where the test was true and one where the test was false, which are then recursed on in the subtrees of the decision node. Each subtree will be either a single leaf node with a label (the final decision) or another subtree where further decisions will be made over the data.

One of the most significant advantages of decision trees is that they are highly interpretable – series of Boolean decision are very intuitive for humans to understand and explain. Consequentially, they are often an appropriate model choice when working in fields where accountability or comprehensibility are required, such as in legal or medical fields.

A. An Example Decision Tree: Swimming

For ease of understanding, we shall now consider an example decision tree. We shall imagine that we are predicting whether or not the swimming pool is a nice temperature for swimming, and we have the following attributes in our input vector to make our decisions:

1. Season ∈ {spring, summer, autumn, winter}
2. Sky ∈ {clear, cloudy}
3. Time ∈ {day, night}

Our target label is Nice ∈ {yes, no}.

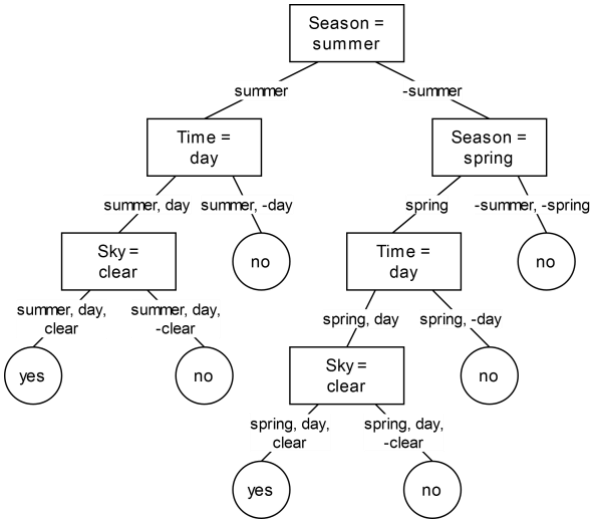

Fig. 1. An example decision tree. The edges note what properties hold on each path, with negation being represented by a dash -. Redundancy is remove for readability.

An individual, when asked when they might want to swim, may say something like "a sunny summer day". In a decision tree, this concept could be represented by a series of decisions such that Season = summer, Sky = clear, Time = day entails Nice = yes. Of course, this is only one possible case. There are other scenarios where it may be nice to swim, and it may not always be nice to swim on sunny summer days – there may be a freak cold front on an otherwise nice day.

In fig. 1, we can see an example decision tree for this problem. We immediately note that the decision tree is not of uniform height – the lowest leaves are at depth 4, and the highest are at depth 2. Intuitively, the higher-level leaves represent decisions that require little information to determine – in this case, if it's the wrong season or not day. Not all decision trees are equal in quality, however, even if they represent the same decisions. In fig. 1, we can see that two assignments lead to Nice = yes:

1. Season = summer, Time = day, Sky = clear
2. Season = spring, Time = day, Sky = clear

All other cases lead to false. Intuitively, we can create a better decision tree that can more quickly eliminate days that are not nice.

The decision tree in fig. 2 eliminates redundant Time and Sky tests and is clearly a more desirable tree. Generally speaking, we are searching for the smallest consistent tree for our training data, but finding the exact best tree is an intractable problem [1]. We shall now discuss how trees are learned.

B. *Learning Decision Trees*

Given that it is not feasible to search the entire space of candidate decision trees, we instead perform constrained searches via some form of greedy selection. Generally, the decision tree algorithm proceeds through the following steps, though specific details and implementations may vary:

1. Test if the data is appropriate to mark as a single class – if so, return a leaf with the appropriate class

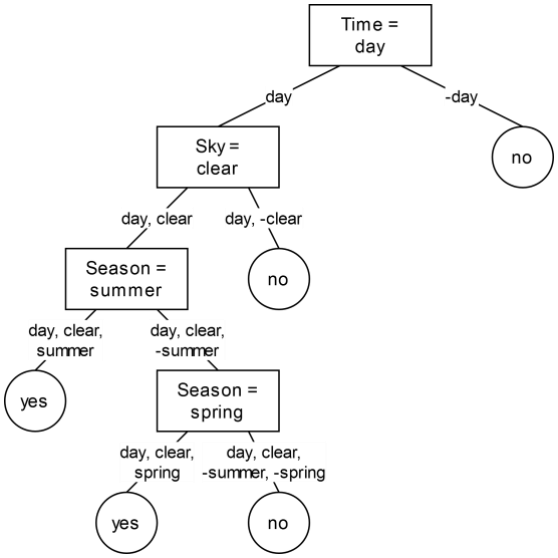

Fig. 2. An improved decision tree

2. Generate candidate decisions for the input data – if there are none, we return a leaf with the majority class
3. Select the best candidate decision
4. Split the data into two sets with the chosen decision
5. Return the internal node as the decision, with the results of recursing on the partition of the input data as its children

In this case, the training data consists of a vector of attribute values and a class. Now, we shall examine each of these steps in brief.

1) Testing for single label: When the data consists of only a single label, there are no further decisions to be made. In practice, some noise is typically allowed for – if the data is overwhelmingly a single label, we return that label. Additionally, if the data is empty (which is generally an indicator that a poor decision was selected), we return a label of the plurality class of the data before the previous split.

2) Generating candidate decsisions: We generate some number of candidate decisions – it is often not feasible to consider all possible decisions – such that the decision are not redundant with those made previously in the tree. If there are no feasible candidates, we must return a label, which will typically be done by selecting the simple majority label of the data. Typically, these checks will be equality or inequality tests on a single attributes, though more sophisticated methods may allow for tests such as "is an element of" or multi-attribute tests. Alternatively, for discrete attributes, the tree may simply split on an attribute into the subchildren where each has a different value for the given attribute – this is illustrated in our experiments.

3) Selecting the best decision: Based on some metric, typically a greedy one, we can then identify the best decision. Here, a common metric is maxizing information gain – whichever decision leads to the greatest reduction in weighted entropy in the produced partition will be selected.

4) Splitting the data: With the best decision selected, the data can be split into two sets – those that are true for the decision, and those that are false.

5) Returning: We construct a new node representing the decision. We recursively define its subchildren by repeating this process on the two sets produced in step 4.

C. Selecting the Best Decision

Given that it is not feasible to exhaustively search the exponentially large decision tree space for the optimal tree, we instead greedily select locally optimal decisions. Generally, we want to select the decision that gives us the most information about the dataset – this can intuitively be understood as the most informative question that an expert may ask about the situation. In our swimming example, this would likely be "Is it daytime?".

In machine terms, this can be expressed as locally optimizing a metric called **information gain**. Information gain is equivalent to **entropy** reduction, where entropy is a measure of chaos in the dataset. Higher entropy corresponds to more uncertain data. Entropy of a random variable V that takes values $v_1 \ldots, v_k$ is defined as:

$$\mathbf{H}(V) = -\sum_{i=1}^{k} \mathbf{P}(v_i)\,\mathbf{log_2}\,\mathbf{P}(v_i)$$

When computing the entropy of a dataset, the values correspond to the distinct elements of the dataset and their probabilities correspond to the frequencies of each element in the dataset.

Information gain can then be expressed as the reduction in entropy from some initial dataset S after being partitioned into sets $S^D{}_1, \ldots, S^D{}_k$ by decision D (in the simple case where only Boolean decisions are permitted, $k = 2$). Note that it is necessary to weight the entropy of each partition of the initial set.

$$\mathbf{IG}(S \mid D) = \mathbf{H}(S) - \sum_{i=1}^{k} \frac{|S_i^D|}{|S|}\mathbf{H}(S_i^D)$$

In order to select the best decision, we simply compute the information gain for each candidate decision and select an arbitrary decision with maximal information gain.

Methods other than information gain, as well as more sophisticated extensions to simply selecting maximal information gain do exist but are beyond the scope of this report.

D. Overfitting

Greedily constructing a decision tree for some input dataset can lead to the common machine learning problem of overfitting, where generalization performance suffers as a consequence of the model being fit for the specific training data given to it. In a decision tree context, we combat overfitting by **pruning** the decision tree learned from the training data [1].

Typically, after learning the tree, we examine internal nodes with only leaves as children. For each node such, we can perform statistical tests to determine if the decision is relevant or is a response to the noise in the data. If it is relevant, we keep the node. Otherwise, we merge the internal vertex and its leaves into a single leaf node predicting the plurality class.

E. Continuous Attributes

In real-world data, it is common for attributes to include continuous or integer-valued data. In decision trees, this scenario is typically handled in one of two ways: **discretization**, where the data is converted to discrete values by some deterministic process (for example, age may be converted into demographic groups such as <18, 18-25, 26-40, 41-65, and 66+); and **splitting**, where split values are chosen algorithmically based upon the input data and the test will take the form of an inequality test such as *Temperature* < 50. Though efficient algorithms exist for selecting efficient split points, it is in practice one of the most computationally expensive parts of the decision tree algorithm. At this point, we're using all attributes containing discrete data and excluded continuous or categorical variables such as Name, Age, Ticket, Fare, and Cabin out of the training process.

F. Distributed Decision Trees

In a world of big data, the decision tree algorithm will lack efficacy if there is no way to efficiently distribute the data-depending processing required. We shall discuss each step and potential areas for distribution, focusing only on a pure decision tree – we will not consider other methods that involve decision trees, such as distributed random forests and boosted methods.

1) Testing for single label: Determining if a node should be a leaf requires examining the number of data points at the node as well as their distribution of labels. A key to optimizing this step is avoiding repeating calculation – when selecting decisions, much of the information required for this step is computed, and can be passed down to the next recursive step as an argument.

2) Generating candidate decsisions: This step can be extremely expensive, especially when continuous values are involved. For discrete values, naively finding distinct options at each step can be prohibitively expensive, so a more sophisticated method of tracking remaining options and passing candidates to the subtrees during recursion is effective. However, sorting and selecting split points for continuous values is much more difficult to optimize, making a single initial discretization an attractive option.

3) Selecting the best decision: The key to distributing decision selection lies in choosing an appropriate metric and avoiding repeated work. Methods such as information gain can work on distributed filter and count operations, and the total count of data points at a given node can be passed to subchildren after recursion. Additionally, intelligent caching and filter optimization can enhance performance when dealing with high-depth trees or more sophisticated classes of decisions.

4) Splitting the data: The key to efficiently splitting the data is to avoid repeating work done while selecting the decision – as the data was already partitioned while determining the best decision, one should simply avoid repartioning, possibly by exploiting cached methods –

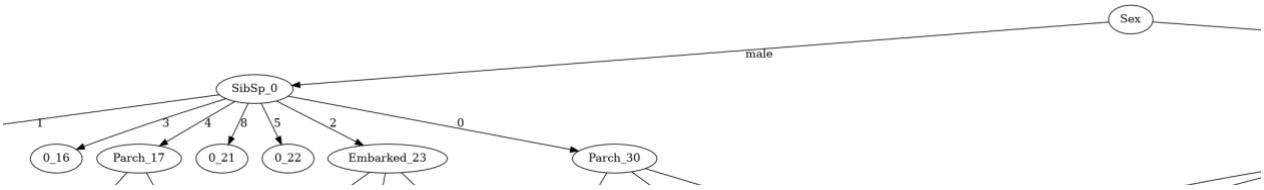

Fig. 3. Part of the left side of the output decision tree.

Fig. 4. Part of the right side of the output decision tree.

though excessive use of caching in trees can easily lead to memory errors.

5) Returning: When defining the node, one should take care to pass as much relevant information to its children as possible, minizing future work. Much of the possible information that can be passed to its children are discussed in steps 1 through 4.

IV. Results and Analysis

We conducted a decision tree computation on the Titanic Kaggle data set. The data set itself consists of passenger details and the primary attribute of survival. Our initial analysis indicated the following attributes to be the most meaningful: PassengerId, Survived, Sex, SibSp (number of siblings or spouses), Parch (number of parents or children), and Embarked.

The method achieved **~81%** accuracy on the training data and **~76%** accuracy on the test data. The variance may be due to noise in the data, or the training tree may be slightly overfit – due to the reduced number of training data from 891 to 183 from dropping all records with missing fields and the small number of attributes, post-pruning would not be productive.

We shall now consider the attributes that indicate survivorship. Clearly, PassengerId serves only as an identifier, and has no predictive value. The other three attributes (Sex, SibSp and Parch) are more related to familial and biological relations. Thus, we infer the presence of familial relations and connect those to the survival rate of the Titanic's passengers.

V. Conclusion and Future Work

We presented a discussion on the theory of decision trees and an example of their use on the Titanic dataset. We produced a decision tree using a distributed PySpark system that achieved reasonable accuracy on the test and train datasets.

An interesting direction for further experimentation would be to compare results with differing decision selection and generation methods, as well as graph-theoretic post-processing to minimize the trees while maintaining consistency. Additionally, if given more resources, it would be interesting to experiment on a larger dataset with more varied attribute types in order to determine the costs of various types of non-discrete attributes, as well as the benefits of different methods of handling non-discrete attributes. Furthermore, an additional technique could be to experiment with missing value imputation to fill in the gaps in missing attributes to produce a more complete dataset. As of this point, we're merely removing all records with missing values, in which significantly reduced the size of our training dataset and could cause overfitting. For example, the method of using linear regression to predict the missing value using target data could be beneficial to fill in large quantities of missing continuous data in many scenarios.

References

[1] S. J. Russell, *Artificial Intelligence:A Modern Approach*, 3rd ed. Harlow: Pearson, 2016.

[2] SF OpenData, "San Francisco Crime Classification," *Kaggle*, 2015. [Online]. Available: https://www.kaggle.com/c/sf-crime/overview. [Accessed: 26-Oct-2020].

[3] Kaggle, "Titanic: Machine Learning from Disaster," *Kaggle*, 2020. [Online]. Available: https://www.kaggle.com/alexisbcook/titanic-tutorial. [Accessed: 22-Nov-2020].