Restaurant Recommendation Challenge

Making prediction by Hadoop and Spark

Bohua Jia
University of Stavanger, Norway
261858@stud.uis.no

Dipanjan Banik
University of Stavanger, Norway
261860@stud.uis.no

ABSTRACT

Prediction is an indispensable part of people's lives, ranging from weather forecasts to national development prospects. A decision tree is an entry-level mechanics algorithm that, while simple and easy to understand, can make very complex and accurate Forecast. Google used decision trees to make market forecasts around 2014 and made money for doing so. This article will present some core functions for the decision tree we implemented using Python. Moreover, using Hadoop mrjob and Spark to handle large datasets. We aim to make a performance comparison of the different data frames.

KEYWORDS

Decision Tree, MrJob, Spark

1 INTRODUCTION

Overview. Enjoying local cuisine could be one of the driving forces behind tourism, and providing travellers with information on local cuisine and restaurant recommendations could be beneficial to them when determining where to go. They are the sorting and searching functions that assist travellers in locating the best local cuisine and restaurants among the vast amount of information available. They can also assist tourists in deciding where to travel.

Research Problem. In today's modern world, we check the ratings of a restaurant before planning to dine. Today, people are more aware than ever of the link between health and nutrition and are more cognizant of what foods they eat when they dine at restaurants. Another modern trend is local foods, with a special emphasis on sustainability. So it is very important to improve the ratings or recommend a restaurant to a customer. Restaurant marketing is the process of getting people to visit a restaurant. Restaurant marketing creates loyalty, provides data for research and analytics, and allows restaurants to better understand their ideal customer profile. As a result, restaurant marketing is extremely important: it allows us to attract new consumers, increase our client base, and turn existing drop-in customers into regulars. Furthermore, the COVID-19 epidemic has emphasized the importance of this even more.

Contribution summary. Using the restaurant recommendation dataset as a starting point, we deployed Machine Learning techniques. We created our own Decision Tree algorithm from scratch and utilized it in the MapReduce and Spark framework to achieve our goals. In addition, we used the sklearn and Spark decision trees to forecast suggestions for a consumer or a vendor to employ. Specifically, Hadoop and Spark are data processing frameworks built to deal with large amounts of data. After that, we looked at the algorithms' performance and demonstrated that our implementation is scalable, and we came to the same conclusion.

The source of or project is available here:
https://github.com/ProJBH/DAT500_project_mrjob_decision_tree

DAT500 - Data Intensive System,
2022.

2 BACKGROUND

2.1 Hadoop

Hadoop [1][2] is a distributed file system and framework that uses the MapReduce paradigm to analyze and convert very large data collections.[3] The most important feature of Hadoop is that it can distribute data to different hosts in the same cluster, thereby greatly improving work efficiency. A Hadoop cluster can grow processing ability, storage capacity, and IO throughput by simply adding commodity servers. In other words, the more servers in the Hadoop cluster, the more powerful it has.

2.2 HDFS

HDFS (Hadoop distributed file system) separates file system metadata from application data. The NameNode, a dedicated server in HDFS, holds system metadata. DataNodes are additional servers that store application data. All servers are ultimately linked and use TCP-based protocols to communicate with others.

2.2.1 *How does it works.*

In HDFS, the file's content is divided into large chunks, and each chunk is replicated at several DataNodes separately. The NameNode is responsible for keeping track of the namespace tree and mapping file chunks to DataNodes (the physical location of file data). When an HDFS client wants to read a file, it first asks the NameNode for the locations of the data chunks that make up the file and then reads the contents of those chunks from the DataNode nearest to the client. The client asks the NameNode to choose three DataNodes to host the block copies when writing data. The client then uses a pipeline to write data to the DataNodes. Each cluster has a single NameNode in the present configuration. As each DataNode can perform many application activities parallelly. So, the cluster can include thousands of DataNodes and ten times more clients than Datanodes.

2.3 Spark

An open-source parallel processing framework that allows in-memory processing to speed up large-scale data analysis applications. Spark is implemented in Scala [4][5]. Unlike Hadoop, Spark is capable of storing intermediate data in memory.[6]

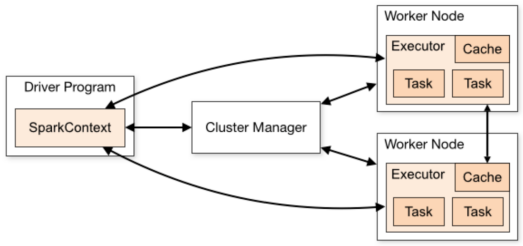

Figure 1: Apache Spark architecture

2.3.1 Driver.

The driver consists of a Spark session. The Spark session takes the task and divides it into smaller tasks that are handled by the worker nodes.

2.3.2 Worker Nodes.

Each worker node receives a task distributed by the driver and executes the task. Those worker nodes are known as a cluster.

2.3.3 Cluster Manager.

The cluster manager communicates with both the driver and the worker nodes to: manage resource allocation, manage program division and manage program execution [7].

2.4 Spark MLlib

Spark MLlib is a Spark Core package that offers machine learning algorithms and MLlib is built for ease of use, scalability, and interaction with other tools. It includes scalable and quick implementations of standard machine learning models. Spark Machine Learning library MLlib contains several different applications like - Regression for Predictions, Clustering, Classification Algorithms, Dimensionality Reduction, and Regression for Predictions. Spark's base computing foundation is a significant benefit. Working on a large-scale machine learning project becomes much easier due to this [9, 10].

2.5 Scikit Learn library (sklearn)

Scikit Learn library has a module function DecisionTreeClassifier() for easily implementing decision tree classifier. In our project, we used the decision tree from sklearn to compare the prediction result.

2.6 Pandas and Spark Datafram

Dataframes contain a table of data with rows and columns. However, Spark Dataframe and Pandas Dataframe are not the same things. Pandas is an open-source Python library based on the NumPy library. It is a Python package that lets us handle numerical data and time series using various data structures and operations. Its primary goal is to make data import and analysis easier. Pandas DataFrame is a two-dimensional, size-mutable tabular data format with labelled axes that might be different (rows and columns). So there are the three fundamental components of a Pandas DataFrame: data, rows, and columns [21].

Spark engine designed for large-scale data processing. It outperforms other cluster computing systems in terms of speed, such as Hadoop. In addition, Spark has several high-level APIs so that it is simple to write parallel jobs in Spark. In Spark, DataFrames have distributed data sets that are organized into rows and columns. Each column in a DataFrame is given a name and a type [21].

When we need to manipulate large volumes of data, we utilize Spark. There are several nodes in a Spark DataFrame. Unlike Pandas, Spark DataFrame ensures fault tolerance. Spark DataFrame is also immutable. Furthermore, Spark DataFrame operations are quicker than pandas since they run parallel on all computers' cores.

2.7 Related works

2.7.1 Supervised learning.

The machine learning task of learning a function that translates an input to an output based on example input-output pairs is known as supervised learning. [11] It uses labeled training data and a collection of training examples to infer a function.[12] Each example in supervised learning is a pair of an input item and the desired output value. A supervised learning algorithm examines the training data and generates an inferred function that can be applied to new cases. The algorithm will be able to accurately determine the class labels for unseen examples in the best-case scenario. This necessitates the learning algorithm to "reasonably" generalize from the training data to unknown conditions.

2.7.2 Random forest.

Random Forest is a computationally efficient technique that can operate quickly over large datasets.[13] Random decision forests are also known as random forests. It is an ensemble learning approach for classification, regression, and other problems that work by training a large number of decision trees. For classification tasks, the random forest's output is the class chosen by the majority of trees. For regression tasks, the mean or average prediction of the individual trees is returned.[14][15] Random forests correct for decision trees' habit of overfitting to their training set.[16] Random forests generally produce better results than decision trees. However, data characteristics can affect their performance.[17][18]

3 RELEVANT ALGORITHMS

3.1 Decision Tree

A decision tree is a form of supervised learning algorithm that may be used to solve issues in both regression and classification. The algorithm creates rules using training data that may represent a tree structure. Like any other tree representation, it has a root node, internal nodes, and leaf nodes. For example, the condition on attributes is represented by the internal node, the branches represent the condition's outcomes, and the leaf node represents the class label. Following the if-else style principles, we can start at the top with the root node and work down to the leaf node to obtain the classification. The class label for our classification challenge is the leaf node where we stop. Both category and numerical data can be used in a decision tree. On the other hand, other machine learning methods cannot deal with categorical input and require encoding to numeric values[20]

One of the predictive modeling methods used in machine learning is decision trees. Utilizing a Decision Tree aims to build a training model that can predict the class or value of a target variable by learning basic decision rules from training data. In Decision Trees,

we begin at the tree's root to forecast a class label for a record. The values of the root attribute are compared to the values of the record's attribute. We follow the branch corresponding to that value and proceed to the next node based on the comparison[7, 8].

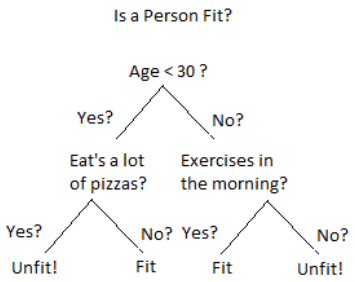

Figure 2: Decision Tree[8]

Classification trees and Regression trees are the two primary forms of Decision Trees that we can create. The fundamental problem in a Decision Tree is determining the attribute for each level's root node, which is known as attribute selection. The Gini index and Information gain are the two most well-known attributes.

The entropy varies as we use a node in a decision tree to split the training examples into smaller groups, and information gain measures this change. Entropy is a measurement of a random variable's ambiguity; it describes the impurity of a random collection of instances. The greater the entropy, the more information is contained. On the other hand, the Gini index is a statistic that measures how often a randomly picked element is mistakenly identified. It indicates that elements of a lower Gini index characteristic should be preferred.[19]

In this project, we have implemented a decision tree algorithm that can recommend a vendor to a customer or a customer to a vendor in both ways. We have implemented this algorithm on Hadoop architecture to see how the clustering works in a distributed environment. In mapper, we used our own decision tree, which we built from scratch and spark implementation; we used the spark MLlib to predict the result.

4 IMPLEMENTATION

This section will present the datasets utilized in the research and the implementation of the decision tree in this part. And also outline the difficulties encountered and the decisions that have been made to solve those challenges in the course of work.

4.1 Dataset

The dataset used in this article is Restaurant Recommendation Challenge from kaggle[22]. The dataset contains several tables. However, during the project, only two tables were involved that are train_full and test_full, as these two tables include every information offered by the other tables. The main difference between train_full and test_full is a column called target, which would be the prediction the decision tree will make. Besides the target column, both tables contain 72 columns. The train_full table has 3.1 GB in size, and the test_full table has 902.52 MB.

4.2 Preprocessing

The goal of the preprocessing stage in our project is to fill the missing values in the testing and the training dataset. These two datasets contain various information about customers and vendors. As per our observation, we found that many rows have missing values and for our project, we did not drop any of the columns and rows with missing values. We applied some different techniques to fill the missing values. Also, in preprocessing stage, we had to deal with some datetime column types. At the final preprocessing stage, we changed all the categorical values to numerical so that the implementation of the decision tree could be done correctly and gives the prediction and accuracy more precisely.

4.2.1 *Filling empty values.*

There must be no empty or null value in the dataset to build the decision tree as a prerequisite. So in the first stage, we filled the missing values with two different strategies.For the string datatype column, we filled the values with the most frequent values; for the integer and double type column, we filled the values with median strategies.

```
for dataTypes in sparkDF.dtypes:
    if dataTypes[1] == "string":
        frequentValueinColumnName = \
        sparkDF.freqItems([dataTypes[0]], 0.60)
        frequentValue = frequentValueinColumnName.collect()[0][
            dataTypes[0] + "_freqItems"
        ]
        if len(frequentValue) == 0:
            # if there is no missing value
            # then continue to next logic
            continue
        else:
            # get the most frequent value and
            # replace the null/missing filed with that value
            sparkDF = \
            sparkDF.fillna(str(frequentValue[0]),
                           subset=[dataTypes[0]])
    elif dataTypes[1] == "int":
        # fill the null/missing value with median
        imputer = Imputer(
            inputCols=[dataTypes[0]],
            outputCols=[dataTypes[0]],
        ).setStrategy("median")
        sparkDF = imputer.fit(sparkDF).transform(sparkDF)
    elif dataTypes[1] == "double":
        # fill the null/missing value with median
        imputer = Imputer(
            inputCols=[dataTypes[0]],
            outputCols=[dataTypes[0]],
        ).setStrategy("median")
        sparkDF =
        imputer.fit(sparkDF).transform(sparkDF)
```

The empty datetime values replaced with the 00:00:00 timestamp.

```
import calendar
for eachDay in calendar.day_name:
    eachDay = eachDay.lower()
    sparkDF = sparkDF.fillna(
        value="00:00:00",
        subset=[
            eachDay + "_from_time2",
            eachDay + "_from_time1",
            eachDay + "_to_time2",
            eachDay + "_to_time1",
        ],
    )
```

4.2.2 *Replacing values.*

Also, we replaced some of the column values with different symbols to use for a later stage.

```
# replace vendor_tag column value from , to -
# replace vendor_tag column value from , to -
sparkDF = sparkDF.withColumn(
    "vendor_tag",
    regexp_replace("vendor_tag", ",", "-"))

# replace vendor_tag_name column value from , to -
sparkDF = sparkDF.withColumn(
    "vendor_tag_name",
    regexp_replace("vendor_tag_name", ",", "-"))
```

4.2.3 *Function for counting number of missing cell value.*

We also wrote a function to count the number of empty cell values in a column. However, we did not run it during the program execution because counting empty values in a column of big data set costs much time and is not practical, so we used the above methods to fill the dataset.

```
def countIfColumnisNull(sparkDF, coLumnName):
    checkNumberofEmptyCells = sparkDF.select(
        [count(
            when(
                isnan(c) | col(c).isNull(), c)
                ).alias(c) for c in sparkDF.columns]
    )
    totalEmptyCellCount = \
    checkNumberofEmptyCells.collect()[0][coLumnName]
    return totalEmptyCellCount
```

4.2.4 *Save the output in HDFS.*

After finishing the filling the missing values we saved the output in HDFS directory.

```
sparkDF.coalesce(1).write.format(
    "com.databricks.spark.csv").option(
        "header", "true").mode(
            "overwrite").save(
    "/dis_materials/testfull_filledValues_output"
)
```

4.2.5 *Cleaning.*

After thoroughly filling all missing data, the next step is to massage the dataset. In this step, we only keep the reasonable proper columns, merge and do calculations between related columns to make the data set tidier, but the information is more concentrated. All information from each row value has a unique key, then passed this key-value pair to the reducer.

```
def mapper(self, _, line):
    random.seed(datetime.now())
    key = random.random()
    rows = csv.reader([line])
    for row in rows:
        if row[1] != "gender":
            ...
            create_at_x = row[4].split(' ')[0]
            ...
            sunday_from_time1 = pd.to_datetime(row[30])
            sunday_to_time1 = pd.to_datetime(row[31])
            sunday_from_time2 = pd.to_datetime(row[32])
            sunday_to_time2 = pd.to_datetime(row[33])
            temSun = ((sunday_to_time1-sunday_from_time1)+
            (sunday_to_time2-sunday_from_time2))
            temSun = temSun.total_seconds()/3600
            sunday = str(round(temSun,1)).split(' ')[-1]
            ...
            primary_tags = row[58]
            primary_tags =re.sub(r"[\\\'\"]", '', primary_tags)
            ...
            tem = vendor_tag.split('-')
            vendor_tag_sum = str(len(tem))
            ...
            target = row[72]
            tu = (customer_id,...,target)
            yield (key,tu)

    def reducer(self, key, values):
        yield (key, list(values))
```

The output:

Figure 3: Cleaning the data

4.2.6 *Formatting.*

In this step, we are going to format the previous output. All the keys will be removed, and the rest of the data will be written as a CSV type of file.

```
for line in sys.stdin:
    line = line.strip()
```

```
    value = line.split("\t")
    if len(value)>=2:
        value = ((value[1].lstrip("["))
        .rstrip("])).replace(" ","")
        value = value.replace("\"","")
        print(value)
```

The output:

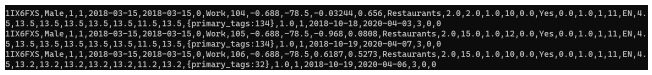

Figure 4: Formatting the data

4.3 Converting categorical data to numerical

Categorical characteristics are string-type data that humans can easily understand. Machines, on the other hand, cannot directly understand categorical data. As a result, categorical data must be converted into numerical data that machines can understand. Machine learning models are unable to understand categorical data. As a result, the conversion to numerical format is required. There are several methods for converting category data to numerical data. The two most famous method is Label Encoding and One Hot Encoding. Our project used the Label Encoding method to convert all the string type features into numerical features. Categorical to numerical conversion is mainly used for the Python sckit-learn and Spark MLlib package to make the decision tree. In Hadoop, we use the Decision Tree, which was built from scratch and it can process categorical features without any conversion. Some categorical features we have reduced to one feature. For example, some columns in both the test and training dataset have a four column value for each day*(sunday_from_time1, sunday_to_time1, sunday_from_time2, sunday_to_time2)*. So we reduced it to only one column *(sunday_opening_time)* and made the value only as a numerical opening hour.

```
for eachDay in calendar.day_name:
    eachDay = eachDay.lower()
    subset = [
        eachDay + "_from_time2",
        eachDay + "_from_time1",
        eachDay + "_to_time2",
        eachDay + "_to_time1",
    ]
    for col_name in subset:
        sparkDF = sparkDF.withColumn(
            col_name, date_format(
                col(
                    col_name), "HH:mm:ss"
                    ).cast("timestamp")
        )
    sparkDF = sparkDF.withColumn(
        eachDay + "_opening_time",
        round(
            (
                unix_timestamp(col(subset[3]))
                - unix_timestamp(col(subset[1]))
                + unix_timestamp(col(subset[2]))
                - unix_timestamp(col(subset[0]))
            )
            / 3600,
            2,
        ),
    )
    for col_name in subset:
        sparkDF = sparkDF.drop(col_name)
```

We have calculated some of the vendors' features manually and converted them into a numerical column.

```
sparkDF = sparkDF.withColumn(
    "vendor_tag", size(
        split(
            col("vendor_tag"), r"\-")))
sparkDF = sparkDF.withColumn(
        "vendor_tag_name", size(
            split(
                col(
                    "vendor_tag_name"
                    ), r"\-"))
    )
```

Finally, we have indexed all the columns with string datatype by the Label Encoding method. The translation of categorical data into numerical data that a computer can interpret is referred to as label encoding. In this system, each type of data is allocated a number beginning with 0. For this purpose, we used the Label Encoder from sklearn to translate these characteristics into numerical values.

```
for dataTypes in sparkDF.dtypes:
    if dataTypes[1] == "string":
        indexer = StringIndexer(
            inputCol=dataTypes[0],
            outputCol=dataTypes[0] + "_indexed"
        )
        sparkDF = indexer.fit(sparkDF).transform(sparkDF)
        sparkDF = sparkDF.drop(col(dataTypes[0]))
        sparkDF = sparkDF.withColumnRenamed(
            dataTypes[0] + "_indexed", dataTypes[0])
```

4.4 Decision Tree

A decision tree is a proper supervised learning technique. It's essentially a categorization issue. It is a diagram in the shape of a tree that depicts a line of activity. It is made up of nodes and leaf nodes. The conclusion is drawn using these nodes and leaf nodes.

4.4.1 *Entropy.*

$$E(s) = \sum_{n=1}^{c} - p_i \ \log_2 \ p_i$$

The entropy of a decision tree measures the purity of the splits. It determines the level of impurity and disorganization. Therefore, it is beneficial to make decisions in a decision tree. For example, it

assists in predicting which node will divide first based on entropy levels.

```
def calculate_entropy(data):
    label_column = data[:, -1]
    _, counts = numpy.unique(label_column,
    return_counts=True)
    pro = counts / counts.sum()
    entropy = sum(pro * -numpy.log2(pro))
    return entropy
```

4.4.2 *Making prediction.*

The prediction is made by applying the row of data with the decision tree.

```
def make_prediction(row, tree):
    question = list(tree.keys())[0]
    feature_name, comparison_operator, value =
    question.split(" ")
    # ask question
    try:
        # feature is continuous
        if comparison_operator == "<=":
            if row[feature_name] <= float(value):
                answer = tree[question][0]
            else:
                answer = tree[question][1]
        # feature is categorical
        else:
            if str(row[feature_name]) == value:
                answer = tree[question][0]
            else:
                answer = tree[question][1]
        # base case
        if not isinstance(answer, dict):
            return answer
        # recursive part
        else:
            residual_tree = answer
            return make_prediction(row,residual_tree)
```

4.4.3 *Calculate accuracy.*

After using the training data to train the decision tree, it is necessary to have a function to test the accuracy of the result.

```
def calculate_accuracy(df, tree):
    df["classification"] = df.apply(make_prediction,
    axis=1, args=(tree,))
    df["classification_correct"] = df["classification"]
    == df["target"]
    accuracy = df["classification_correct"].mean()
    return accuracy
```

4.4.4 *Splitting training dataset.*

Since the test dataset has no target column, the only way to determine whether the predictions are correct is to split the training set into two subsets—one for training the decision tree and another for testing the predictions. In that case, we decided to split the dataset randomly. It can avoid constantly training the tree with a similar dataset if the accuracy is not satisfactory.

```
def train_test_split(df, sub_size):
    if isinstance(sub_size, float):
        test_size = round(sub_size * len(df))
    indices = df.index.tolist()
    test_indices = random.sample(population=indices,
    k=test_size)
    train_df = df.loc[test_indices]
    test_df = df.drop(test_indices)
    return train_df, test_df
```

4.5 Applying Decision Tree algorithm with MapReduce framework

4.5.1 *Splitting.*

Since the data size is too large for a standard python file, splitting the training dataset has to be done by mrjob. We decided to use a random function with a current timestamp as the random seed to produce the random number to indicate whether the current row belongs to the sub-training set or sub-testing set.

```
def mapper(self, _, line):
    random.seed(datetime.now())
    tag = random.random()
    line = line.strip()
    row = line.split(',')
    if row[0] != "customer_id":
        num = random.randint(0,10)
        if num <=1:
            key = 'Test'+str(tag)
            yield (key,row)
        else:
            key = 'Train'+str(tag)
            yield (key,row)

def reducer(self, key, values):
    yield key, list(values)
```

After sub-datasets have been produced, the output again needs to format.

4.5.2 *Formatting.*

The main difference between formatting for sub-datasets and original datasets is when the reading line from the input file, the keys are different. And in this step, we will read the key separately. Since the mrjob cannot produce more than one file once, we must use two similar mappers to handle it and create datasets for sub-training and sub-testing.

```
for line in sys.stdin:
    line = re.sub("[\"\']", "", line)
    #find('Train') for sub_train dataset
    if line.find('Test') == 0:
            line = line.strip()
            value = line.split("\t")
```

```
            if len(value)!=0:value =
                ((value[1].lstrip("[")).rstrip("]"))
                .replace(" ","")
                print(value)
```

4.5.3 *Building Decision Tree.*

This step will build the decision tree with sub-training dataset and using the function implemented from scratch. Since the decision tree cannot be made for every line from the input file, we saved the line's information to a local variable after reading all lines and then used it to train and build the decision tree.

```
import ...
import zipimport

importer = zipimport.zipimporter('DT.mod')
dt = importer.load_module('DecisionTree')
column_name = ['customer_id',...,'target']
# An empty tuple
csv_context = ()
for line in sys.stdin:
    line = line.strip()
    row = line.split(',')
    if 'customer_id' not in str(row):
        added_value_tuple = (row,)
        csv_context = csv_context + added_value_tuple

df = pd.DataFrame(csv_context, columns=column_name)
# change data types
df = df.astype({'sunday': np.float64 ...})
tree = dt.decision_tree_algorithm(df)
print('%s\t%s' % ('tree',tree))
```

4.5.4 *Calculating accuracy.*

This step will test the accuracy of the predictions made by the decision tree. However, before testing the predictions, the output of the building decision tree must merge with the sub-testing dataset. The mrjob does not support taking two input files at once. Once merged, the decision tree can be identified by its key when reading the input file. And then can use the function implemented in the scratch to get accuracy on the tree.

```
importer = zipimport.zipimporter('DT.mod')
dt = importer.load_module('DecisionTree')
column_name = ['customer_id',...,'target']
trees = []
csv_context = ()
for line in sys.stdin:
    if line.find('tree') == 0:
        line = line.strip()
        line = line.split('\t')
        tree = line[1]
        tree = ast.literal_eval(tree)
        trees.append(tree)
    else:
        line = line.strip()
        row = line.split(',')
        if 'customer_id' not in str(row):
            added_value_tuple = (row,)
            csv_context = csv_context + added_value_tuple

df = pd.DataFrame(csv_context, columns=column_name)
df = df.astype({'sunday': np.float64, ...})
for i in range (len(trees)):
    accuracy = dt.calculate_accuracy(df,trees[i])
    print('%s\t%s'%('accuracy',accuracy))
```

4.5.5 *Making predictions.*

If the accuracy of the decision tree is satisfied, then we also need to combine the decision tree output with the testing dataset like in the last step. Finally, call the function implemented in the scratch to make predictions.

```
importer = zipimport.zipimporter('DT.mod')
dt = importer.load_module('DecisionTree')
column_name = ['customer_id',...,'location_number_obj']
trees = []
csv_context = ()
for line in sys.stdin:
    if line.find('tree') == 0:
        line = line.strip()
        line = line.split('\t')
        tree = line[1]
        tree = ast.literal_eval(tree)
        trees.append(tree)
    elif len(trees)!=0:
        line = line.strip()
        row = line.split(',')
        if 'customer_id' not in str(row):
            csv_context = (row,)
            df = pd.DataFrame(csv_context,
            columns=column_name)
            df = df.astype({'sunday': np.float64, ...})
            x = dt.make_prediction(df.iloc[0],trees[0])
            row.append(x)
            print(row)
```

4.6 Applying Decision Tree algorithm using MapReduce framework and Sklearn package

In this session, we build the decision tree using the Sklearn package with mrjob, and it uses the numerical data as input. Unlike the self-implemented decision tree, the decision tree from Sklearn does not require that many preprocessing steps.

4.6.1 *Data sets preparation.*

In the data preparation, we read every input line and give it a unique key using a random number with the current timestamp as a random seed. And then add an identified tag with the key. Then pass every row to the reducer and print the result as the new dataset.

```
class sklearn_train_preparation_map(MRJob):
    def mapper(self, _, line):
```

```
        random.seed(datetime.now())
        key = random.random()
        rows = csv.reader([line])
        for row in rows:
            # + 'test' for testing side
            yield (key+' train',row)

    def reducer(self, key, values):
        yield (key, list(values))
```

After producing both new training and new testing datasets, merge them into one dataset.

4.6.2 *Making prediction.*

When reading data from the input file, the data is split into two tuples based on the different key values—one for training and another for testing. Finally, applying the function from the Sklearn package to build and train the decision tree and use the tree to predict.

```
training_set = ()
testing_set = ()
column_name = ['customer_id',...,'target']

for line in sys.stdin:
    line = line.strip()
    values = line.split('\t')
    key = values[0].split(' ')
    if key == 'train':
        value = values[1].split(',')
        training_set = training_set + (row,)
    elif key == 'test':
        value = values[1].split(',')
        testing_set = testing_set + (row,)

df_train = pd.DataFrame(training_set, columns=column_name)
df_test = pd.DataFrame(testing_set, columns=column_name)
X = df_train.drop("target", axis=1)
y = df_train.target
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.3)
clf = DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)

for i in range(len(df_test))
    test_pred = clf.predict(df_test[i])
    df_test[i].append(test_pred)
    print(df_test[i])
```

We were limited on time and opted to focus on implementation in Spark, hence the mrjob for Sklearn decision tree has not yet been properly debugged.

4.7 Applying Decision Tree algorithm using PySpark and Spark MLlib

PySpark is the Python API for Apache Spark and MLlib is a wrapper over the PySpark and Spark's machine learning (ML) library. In our project we used the DecisionTreeClassifier from the PySpark ml classification package.

4.7.1 *Loading the data.*

First, we initialize the spark session and load the preprocessed data from HDFS. We are now using the preprocessed numerical only data to make the Decision Tree.

```
flieList = {
    0: "hdfs://namenode:9000/test_categoricalToNumerical.csv",
    1: "hdfs://namenode:9000/train_categoricalToNumerical.csv",
}

spark = \
SparkSession.builder.appName(
    "Spark Decision Tree").getOrCreate()
sparkDFTrain = spark.read.csv(
    flieList[1],
    header=True,
    inferSchema=True,
)
sparkDFTest = \
spark.read.csv(
    flieList[0],
    header=True,
    inferSchema=True,
)
```

4.7.2 *Making Decision Tree.*

Before making the Decision Tree, we need to select some of the features and a label column. From our dataset, the label column is the target and except for the target column, we use all the other columns as our features. We used the MLlib VectorAssembler function and VectorAssembler is a feature transformer that merges multiple columns into a vector column. It combines all the feature columns into a vector. After that we selected the features and labels as a model data frame.

```
featureColumns = (
        "customer_id",
        "gender",
        "status_x",
        ...
    )
# converting the feature list into a vector
assembler = VectorAssembler(
    inputCols=featureColumns, outputCol="features")
outputTrain = assembler.transform(sparkDFTrain)
# only features and label for making decision tree
modelDFTrain = outputTrain.select("features", "target")
```

With a max depth of 4, we used two types of impurity methods to make the comparison.

```
dtc = DecisionTreeClassifier(
        labelCol="target",
        featuresCol="features",
```

```
        maxBins=5000,
        maxDepth=4,
        # impurity="gini"
        impurity="entropy",)
```

4.7.3 Make prediction and calculate accuracy.

After making the decision tree object, we did the same task for our testing set and separated the features and label. First, we fit the training dataset into the Decision Tree Classifier (dtc) model and then predict by transforming the model's test set. MulticlassClassificationEvaluator function gives us the final accuracy result which we will discuss in the next section.

```
dtcModel = dtc.fit(modelDFTrain)
modelDFTest = outputTest.select("features", "target")
dtcPred = dtcModel.transform(modelDFTest)
evaluator = MulticlassClassificationEvaluator(
    labelCol="target",
    predictionCol="prediction",
    metricName="accuracy")
dtcAcc = evaluator.evaluate(dtcPred)
print("accuracy", dtcAcc)
```

5 EXPERIMENTAL EVALUATION

In this chapter, we will analyze the outcomes of the algorithms constructed and applied to our dataset. We will talk about the probable weaknesses of our implementations and some approaches to make them better moving forward. The algorithms running time will also be described, and their performance will be evaluated as a function of changes in the hardware configurations and algorithm parameters. We will also analyze the outcomes of the decision tree algorithms with MrJob and Spark on the dataset. We will also talk about the potential flaws in our implementations and how we can fix them.

5.1 Cluster Setup

Our cluster configuration consists of a network of four virtual machines (VMs), each of which has 4GB of RAM and two core CPUs. Four virtual machines (VMs) have been setup as a single cluster consisting of one master node and three subordinate nodes. The Hadoop 3.2.1 distribution has been deployed on the cluster, and Yarn is being used as the resource manager. The version of Spark that has been installed is 3.1.3. Yarn is also used as a cluster manager for Spark tasks, which we find to be quite useful. All of the testing and debugging were made in namenode and.

5.2 Performance evaluation of Spark

We observed that the Spark job runs faster than Hadoop during the experiment. We also measured the execution time and accuracy between two methods of Spark Decision Tree and Hadoop MapReduce. Our experiment shows that when we use the node impurity as Gini full data set gives lower accuracy than node impurity as Entropy.

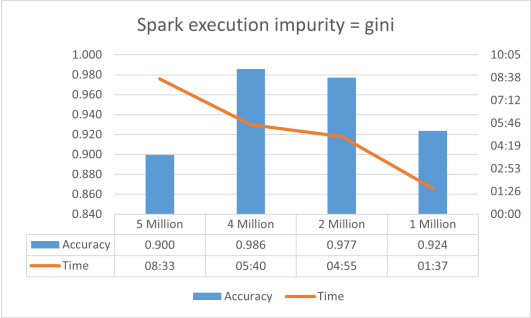

Figure 5: Spark execution comparison 1

Also we observed that time consumption is also sightly high in entropy. As we reduce the dataset sample size, accuracy also increases and execution time decreases. However, when we used the data chunk size of 2million, the result is pretty much the same in both Gini and Entropy impurity.

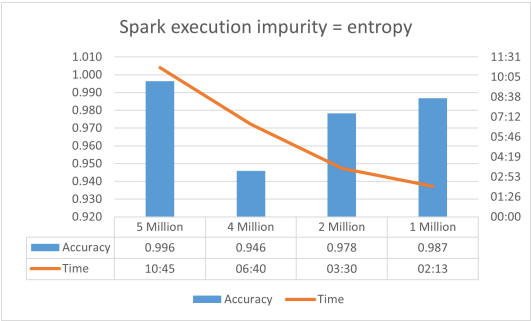

Figure 6: Spark execution comparison 2

5.3 Performance evaluation of Hadoop

Unlike Spark Hadoop execution time takes much time, our observation implies that the accuracy result remains the same even if we test with smaller chunks of data. On a full dataset, Hadoop execution time takes about 50min to complete and gives the output, and it is five times slower than Spark maximum execution time (11min) compared with Spark. Nevertheless, as we can see that accuracy remains constant, we can imply that Hadoop may be used where the data prediction needs to be more precise.

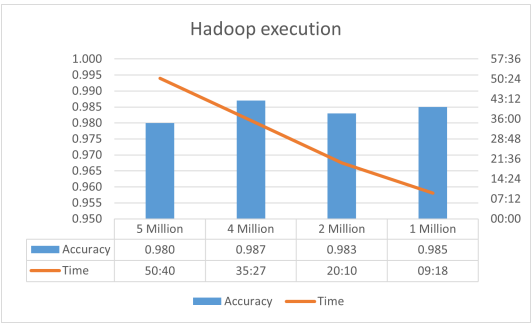

Figure 7: Hadoop execusion analysis

5.4 Challenges

5.4.1 *Converting the mrjob output.*

The output of mrjob is plain text, so the first thing we need to solve is converting the output of mrjob into the correct format or the correct data architecture. We encounter this kind of problem at the very beginning. Then we develop formatting mrjob to overcome it. Even when we need to pass the plain text as a decision tree which builds from scratch, we find a way to read the text as the dictionary type, but when we try to pass the text as a decision tree object for the Sklearn package, we do not find any possible way to that. So, instead of passing the tree object from different mrjob, we decided to build the tree when all data have been read. Furthermore, we think that may cause a problem when using mrjob since the data may overflow the RAM, but that will not be a problem for Spark. So, we think the improvement can be made by shifting the mrjob to Spark in the future.

5.4.2 *Importing third party packages.*

We encounter a problem when importing the Numpy and Pandas package. Then we figure it out by using pip install relative packages. However, we also implement a decision tree from scratch, but we cannot solve the problem with previous experience. So finally, we find a way to zip the original class, rename the extension from .zip to .mod first, and use zipimporter to load the module.

5.4.3 *Randomly split the dataset.*

That was not a problem until we realized that once the decision tree is built with a lower accuracy rate, it will never be satisfied. Then we came up with randomly splitting the dataset and using different data to train the decision tree. So finally, this problem has been solved.

6 CONCLUSION

In this project, we used mrjob to create a Machine Learning algorithm, namely Decision Tree. We used an Apache Hadoop cluster as the working environment to apply the method to a collection of data with varying data sizes.

We compared the results provided on the same dataset using well-designed and efficient libraries such as Spark's mllib, Python's sklearn, and the self-built decision tree module to the results produced by the implemented method. Finally, we show how the prediction results compare using a bar chart. Our decision tree's predictions are not exceptionally efficient but comparable with other libraries.

For future work, we have opportunities to improve the performance of building the decision tree more efficiently. Furthermore, have the opportunities to find a better way to manage the stored data to avoid memory overflow.

REFERENCES

[1] Apache hadoop,https://hadoop.apache.org/
[2] J. Venner, Pro Hadoop, Apress, June 2009.
[3] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system[C]//2010 IEEE 26th symposium on mass storage systems and technologies (MSST). Ieee, 2010: 1-10.
[4] Scala programming language. http://www.scala-lang.org.
[5] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[C]//2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10). 2010.
[6] Tomasz Wiktorski. 2019. Data-intensive Systems: Principles and Fundamentals using Hadoop and Spark. https://doi.org/10.1007/978-3-030-04603-32
[7] [n. d.].Decision Trees in Machine,Learning.https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052
[8] Decision Trees for Classification, https://www.xoriant.com/blog/product-engineering/decision-trees-machine-learning-algorithm.html
[9] What is Machine Learning Library (MLlib) | Databricks,https://databricks.com/glossary/what-is-machine-learning-library
[10] [n. d.].Spark MLlib for Scalable Machine Learning with Spark.https://www.projectpro.io/article/spark-mllib-for-scalable-machine-learning-with-spark/339mcetoc_1fb3o9fds8
[11] Stuart J. Russell, Peter Norvig (2010) Artificial Intelligence: A Modern Approach, Third Edition, Prentice Hall ISBN 9780136042594.
[12] Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar (2012) Foundations of Machine Learning, The MIT Press ISBN 9780262018258.
[13] Oshiro T M, Perez P S, Baranauskas J A. How many trees in a random forest?[C]//International workshop on machine learning and data mining in pattern recognition. Springer, Berlin, Heidelberg, 2012: 154-168.
[14] Ho, Tin Kam (1995). Random Decision Forests (PDF). Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14–16 August 1995. pp. 278–282. Archived from the original (PDF) on 17 April 2016. Retrieved 5 June 2016.
[15] Ho TK (1998). "The Random Subspace Method for Constructing Decision Forests" (PDF). IEEE Transactions on Pattern Analysis and Machine Intelligence. 20 (8): 832–844. doi:10.1109/34.709601.
[16] Hastie, Trevor; Tibshirani, Robert; Friedman, Jerome (2008). The Elements of Statistical Learning (2nd ed.). Springer. ISBN 0-387-95284-5.
[17] Piryonesi S. Madeh; El-Diraby Tamer E. (2020-06-01). "Role of Data Analytics in Infrastructure Asset Management: Overcoming Data Size and Quality Problems". Journal of Transportation Engineering, Part B: Pavements. 146 (2): 04020022. doi:10.1061/JPEODX.0000175. S2CID 216485629.
[18] Piryonesi, S. Madeh; El-Diraby, Tamer E. (2021-02-01). "Using Machine Learning to Examine Impact of Type of Performance Indicator on Flexible Pavement Deterioration Modeling". Journal of Infrastructure Systems. 27 (2): 04021005. doi:10.1061/(ASCE)IS.1943-555X.0000602. ISSN 1076-0342. S2CID 233550030.
[19] [n. d.].Decision tree introduction with example.https://www.geeksforgeeks.org/decision-tree-introduction-example/
[20] [n. d.].Decision Tree Classifier in Python Sklearn with Example.https://machinelearningknowledge.ai/decision-tree-classifier-in-python-sklearn-with-example/
[21] [n. d.].Difference Between Spark DataFrame and Pandas DataFrame.https://www.geeksforgeeks.org/difference-between-spark-dataframe-and-pandas-dataframe/
[22] ANDRII.2022.Restaurant Recommendation Challenge.https://www.kaggle.com/datasets/mrmorj/resta recommendation-challenge