A project report on

Automated Univariate Analysis And Decision Tree Formulation

Submitted in partial fulfillment for the award of the degree of

Integrated Mtech.(Software Engineering)

by

Neeladri Chatterjee(17MIS7137)

AMARAVATI

SCOPE,VIT-AP

July,2020

Tatvic Analytics Private Limited
Corporate Office:
Office No. 402 & 403,
4th Floor, Camps Corner II,
100 ft Road, Prahaladnagar,
Ahmedabad – 380015, Gujarat.

29th May 2020

Internship at Tatvic

Dear Neeladri,

With reference to your internship from 5th April 2020 to 29th May 2020 we take great pleasure in informing that you have been completed your internship project with Tatvic Analytics Pvt. Ltd.

We wish you all the best with your future endeavours.

With best wishes & warm regards,
For Tatvic Analytics Private Limited:

Zil Rindani
Talent Acquisition Analyst

ACKNOWLEDGEMENT

It is my pleasure to express with deep sense of gratitude to Prof. Asish Dalai, VIT-AP, for his/her constant guidance, continual encouragement, understanding; more than all, he taught me patience in my endeavor. My association with him / her is not confined to academics only, but it is a great opportunity on my part of work with an intellectual and expert in the field of Data Science.

I would like to express my gratitude to Dr. G.Viswanathan, Dr.Sankar Vishwanathan, Dr. D.Subhakar, and Prof. Jagdish Mudiganti for providing an environment to work in and for his inspiration during the tenure of the course.

In a jubilant mood I express ingeniously my whole-hearted thanks to Dr. D Sumathi, Program Coordinator (MTSE), all teaching staff and members working as limbs of our university for their not-self-centered enthusiasm coupled with timely encouragements showered on me with zeal, which prompted the acquisition of the requisite knowledge to finalize my course study successfully. I would like to thank my parents for their support.

It is indeed a pleasure to thank my friends who persuaded and encouraged me to take up and complete this task. At last but not least, I express my gratitude and appreciation to all those who have helped me directly or indirectly toward the successful completion of this project.

Place: Amaravathi

Date: 12/07/2020 Neeladri Chatterjee

ABSTRACT

The project focuses on extracting the data from a cloud platform in a particular desired format, before running it on the server. The data obtained, is then used to perform univariate analysis and decision tree formulation. After the decision tree formulation, the insights obtained will be used by the client to drive marketing campaigns.

CONTENTS

A.) BIGQUERY AND DATASET CREATION

a.) Explore Python Client library of BigQuery

- Python has several in-built packages and libraries which can be used to achieve several tasks in a simpler and efficient way.
- BigQuery is a google-powered serverless data storing warehouse which can be used to store vast amounts of data at a very cheap cost.
- To run the operations on the dataset on the server, it requires a lot of time, because the server needs to send the request to bigquery to be applied on the dataset multiple times.
- Therefore, we will run the operations on bigquery and get the preprocessed data directly from bigquery.
- The process of fetching data from bigquery requires the use of a python client library.
- The python client library has a lot of other methods also, which can be used to carry out other operations like extracting a particular type of column.

b.) Get column data types from the dataset.

- Each dataset that is used in analytics has two types of columns, categorical and numerical.
- The dataset type is important to know because that will help us give an idea about the data present in it and how to preprocess it.
- To access the data about the column type, INFORMATION_SCHEMA.COLUMNS is used.
- This will help us in identifying which operations are to be carried out on which column of the dataset.

c.) Create a dataset for insights generation

- So, whatever I had learnt under the explore bigquery and python client library task, I had to use that to connect a main function with parameters.
- The parameter in the function is the Query that is to be passed to bigquery using the python client library.
- The data to be fetched from the bigquery should be arriving back at the end of the function in the form of a dataframe.
- This function will be repeated again and again by various functions for getting the data from bigquery.
- The single function reduces the lines of code that could have been redundant.

d.) Create additional fields from VisitStartTime from BigQuery

- The dataset in Bigquery has the time in UTC format (time in seconds from a particular date till the time that observation was recorded).
- The UTC-time is converted to normal time format.
- Then, various other column data were extracted from the normal date and time format column.
- The different data that were extracted are:
    - Month.
    - Day of the week.
    - Day of the month.
    - Week of the year.
- This will help in making the predictions and classifying the different types of user more precisely.

B.) UNIVARIATE ANALYSIS

a.) Dynamic Bucket Size Selection For Numeric

- The data that is present in a numeric column is really hard to analyse, since there can be a large number of discrete values present.
- In order to deal with this problem, we had to make different buckets for the columns and count the number of values coming in each column, along with coverage of each bucket generated.
- The mean of the column is calculated and a z-score of 2 is allowed, for data to be considered while bucket formation.
- The lower limit and upper limit for bucket formation are set by subtracting the z-score from the mean values.
- Two more buckets are made to consider the values lying outside the lower and upper limit points for bucket formation.
- The SQL CASE operation is used to classify different discrete points in various buckets.

b.) Basic Overview of Distribution For Numeric

- All the data points for numeric will be classified into bins using SQL CASES and the count will also be generated.
- After calculating the count, the coverage of each basket is to be calculated and sorted to give an idea to the user regarding the basket with the highest and the least coverage from the data set.

c.) Top Count And Coverage For Categorical Data

- In the categorical columns,the count for each discrete value was to be calculated and a dataframe with the count for each discrete value should have come in the output.
- The coverage is also to be calculated for the discrete values and then it should be sorted in decreasing order to get an idea of which discrete value is most important from the categorical column.

d.)Basic Overview of Distribution For Numeric

- A data frame with the discrete value along with count and coverage should be given in the form of output.
- This output will be useful for the user to analyse the column and get the idea of the distribution of the data in the column.

e.) Comparison for converter and non-converter data

- Both the numeric baskets and categorical values along with the count and coverage values are segregated on the basis of whether the lead is converted or non-converted.
- The coverage values were counted for converted and unconverted numeric baskets and categorical values.
- The converted percentage was kept on one side,while unconverted was on the other side with the numeric baskets or categorical values in between depending on the column type.
- Then, the output is given in the form of a data frame.

C.) DECISION TREE

a.) Data Cleaning (Drop Null Values and Repetitive Fields)

- The data present in the numerical columns has a lot of null values and repetitive fields present in them.
- For each column, the count of each distinct value was found and a parameter was given to the function which could be set by the user.
- This parameter acts as a threshold value.
- If the count of any distinct value crosses the threshold, then those values are dropped from the dataset.
- In order to deal with the null values, after removal of the repetitive fields, DROPNA command is used for the data frame to remove any null value present in a column.

b.) Pre-processing(Filling of missing values) of numeric and categorical column

- Since, missing values in the dataset can adversely affect the classification of different values using a decision tree, that's why it is very important to fill in the missing values.
- For numeric columns, all the missing values are to be filled by a very distant number, because filling a close number may hamper the classification process of the decision tree.
- For categorical columns, all the missing values are substituted using mode value because as a particular value is appearing too many times, we can assume that the missing values had the same value as that.
- Pre-processing forms an important part as it can make the classification good or bad on the basis of the method of filling these missing values.

c.) Pre-processing (Top count/Others grouping) of categorical columns

- Since there is a probability that the number of distinct values in a categorical column are too much.
- So, to deal with this situation, a function was made with a threshold to be set by the user, on how many values can be allowed to be exempt from grouping into Other category.
- The count and coverage was calculated for the categorical column and then the column discrete values were arranged on the basis of decreasing order of coverage.
- The values after the threshold value are clubbed together to form the ‘Other’ category.

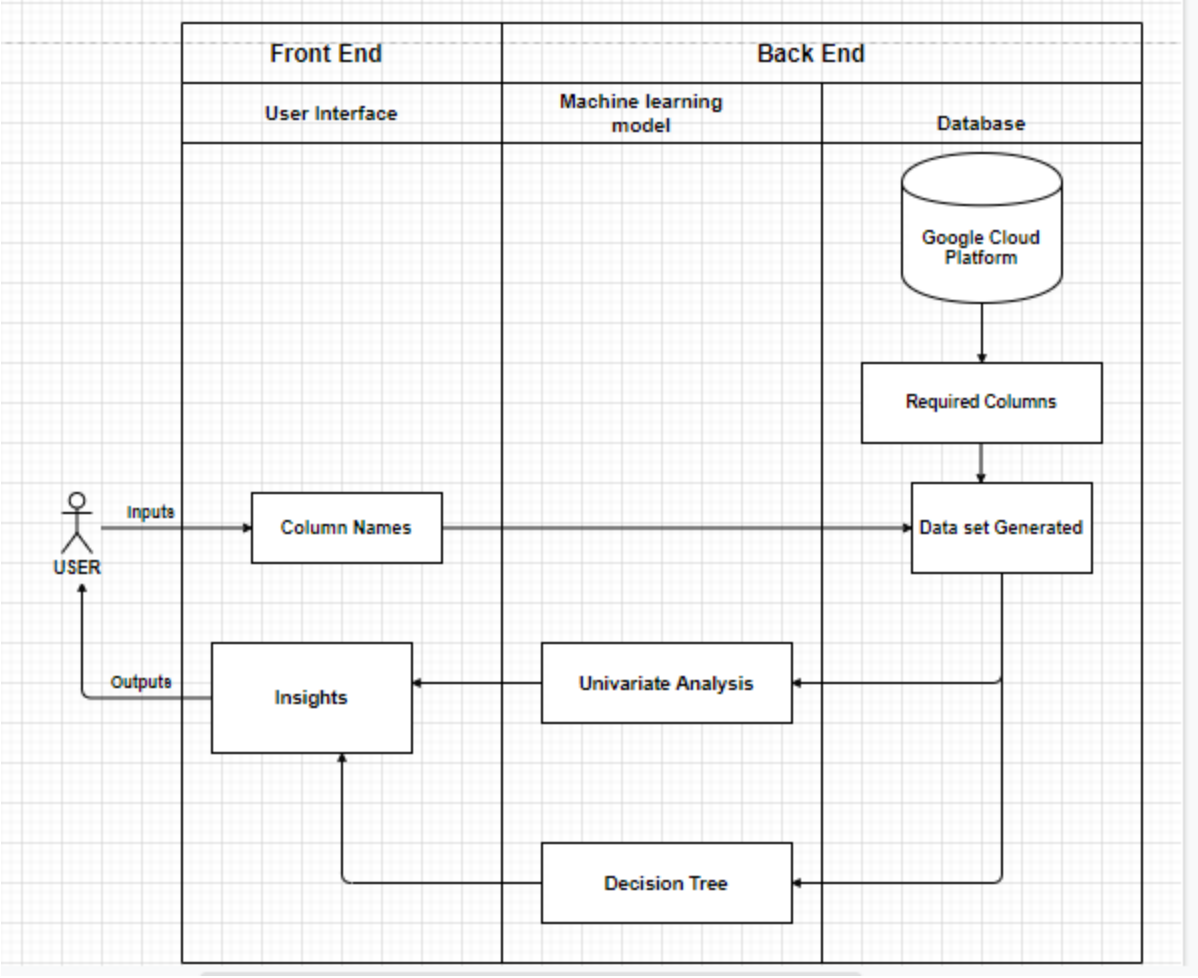
d.) Pre-processing (One hot encoding) of categorical columns

- Since the code is to be run on a computer which cannot understand categorical data present in the dataset, so it becomes very important to convert it into numerical form to be processed.
- We will use pd.dummies on the dataset to create dummy variables for the categorical data present.

e.) Decision Tree Hyperparameter Tuning

- Since we cannot manually tune the model to get the perfect decision tree classification on the dataset.
- We need to run the dataset on a cross validation model which finds out the best hyperparameters on which the data will give the best outputs.
- Two-three hyperparameters are kept in the cross validation model for which the values are saved.
- The model after running on those particular values, is saved on the system using the pickle python library.

PROCESS FLOW DIAGRAM



CODE

A.)Univariate Analysis

```
"""This module contains various functions required
to carry out univariate analysis of a
table after generating it dynamically from bigquery. """
from google.cloud import bigquery
import pandas as pd
def column_info(column_name, project_name, table_name):
    """ This function gives information regarding the datatypes of different columns
        Parameters required:a.)project_name: The project name in which the table is located
                            b.)table_name: Name of the table.
                            c.)column_name: Name of the column.
        Result: A dataframe with all the column names and their datatypes"""
    query = """WITH column_numeric as(
                    SELECT DATA_TYPE,COLUMN_NAME FROM {project_name}.INFORMATION_SCHEMA.COLUMNS
                    WHERE table_name="{table_name}")
                    Select DATA_TYPE FROM column_numeric WHERE COLUMN_NAME = "{col_name}";
                """.format(project_name=project_name, table_name=table_name, col_name=column_name)
    result = main_func(query)
    return result
def dynamic_bucket(column_name, project_name, table_name, buckets=10):
    """This function will automatically make buckets for any numeric column,
    Parameters Required:
        a.)column_name: The name of the numeric column for which you want
                        to make dynamic buckets and get the count and coverage.
        b.)buckets: The number of buckets you want to make.
        c.)project_name: The project name in which the table is located
        d.)table_name: Name of the table.
    Result: A string with dynamic buckets will be made."""
    #This Query will be passed to the main function to get the data for the
    #column name provided after formatting the column name in the string.
    query = """SELECT AVG({col_name}) as Mean, STDDEV({col_name}) as St_deviation,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(0)] AS min,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(100)] AS max
            from
            {project_name}.{table_name}""".format(
```

```
            col_name=column_name, project_name=project_name, table_name=table_name)
    data = main_func(query)
    #converting to float to NaN incase the any value in the column is in string format
    data = data.transpose()
    data = data.reset_index(drop=True)
    #converting dataframe into series for calculating mean
    #standard deviation without dtypes in the output.'''
    data_mean = round(data.iloc[0, :].values[0], 2)
    data_std = round(data.iloc[1, :].values[0], 2)
    #setting the min and max limit for the buckets.
    # taking the range from -2σ to 2σ for removing outliers.
    min_range = round(data_mean-(2*data_std), 2)
    max_range = round(data_mean+(2*data_std), 2)
        #width of bucket considering the range from min_range to max_range.
    bucket_width = (max_range-min_range)/buckets
    #the outliers will be moved to two separate buckets.
    #so the total number of buckets will be two more .
    #than the buckets provided by the user.
    number_of_buckets = buckets+2
    #i is used as an iterator for the number of buckets.
    #k is a constraint put up to stop the bucket creation
    # when the upper limit of a bucket is the same as the max_range.
    i, k = 0, 0
    #This constant is used to declare an empty string
    # in which the values obtained after each iteration will be added.
    query = ""
    for i in range(0, number_of_buckets):
        #creating outlier bucket
        # for values lower than the min_range
        if i == 0:
            text_2 = "WHEN {col_name} < {min_range} THEN '[{min_data}-{min_range})'\n".format(
                col_name=column_name, min_data=str(round(data.iloc[2, :].values[0], 2)),
                min_range=str(round(data.iloc[2, :].values[0], 2)))
            query = query+text_2
        #creating outlier bucket for values higher than the max_range
        elif i == number_of_buckets-1:
            text_3 = "WHEN {col_name} >= {max_range} THEN '[{max_range}-{max_data})'\n".format(
                col_name=column_name, max_data=str(round(data.iloc[3, :].values[0], 2)),
                max_range=str(round(data.iloc[3, :].values[0], 2)))
            query = query+text_3
        #creating dynamic buckets using min_range and max_range.
        elif k != number_of_buckets-2:
```

```
            #increasing the value of k by 1 to change
            # the new lower limit by the previous iteration upper limit
            # and the new upper limit increases by the bucket width.
            lower_limit = round(min_range+(k*bucket_width), 2)
            upper_limit = round(min_range+((k+1)*bucket_width), 2)
            k = k+1
            text_4 = "WHEN {lower_limit}>={column_name} AND {column_name}<{upper_limit} THEN
 '[{lower_limit}-{upper_limit})'\n".format(
                lower_limit=str(lower_limit), upper_limit=str(upper_limit), column_name=column_name)
            query = query+text_4
        i += 1
    return query
def numeric_data_overview(column_name, project_name, table_name):
    """ This function provides the basic overview for a numeric column.
        Parameters to be passed :
            a.)column_name: Name of the numeric column for which you want to see the overview
            b.)project_name: The project name in which the table is located
            c.)table_name: Name of the table.
        Result: You will get a single column matrix with the values Mean,Standard_deviation,
                quantiles(25,50,75),Min,Max.
    """
    # This Query will be passed to the main function
    # for calculation of mean,std_deviation,quantiles,minimum and maximum
    # after formatting.
    query = """SELECT AVG({col_name}) as Mean, STDDEV({col_name}) as St_deviation,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(0)] AS min,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(25)] AS quantile_25,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(50)] AS quantile_50,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(75)]  AS quantile_75,
            APPROX_QUANTILES({col_name}, 100)[OFFSET(100)] AS max
            FROM (SELECT SAFE_CAST({col_name} AS FLOAT64) as {col_name} FROM
            {project_name}.{table_name})""".format(
                col_name=column_name, project_name=project_name, table_name=table_name)
    result = main_func(query)
    #transposing the result to get the answers in a singular column.
    result = result.transpose()
    # renaming the column after transpose with the column name provided in the parameters.
    # 0 is the predefined name given to the singular column after transpose.
    result = result.rename(columns={0:column_name})
    return result
def categorical_overview(column_name, project_name, table_name):
    """ This function provides the basic overview for a categorical column.
```

```
      Parameters to be passed :
           a.)column_name: Name of the numeric column for which you want to see the overview
           b.)project_name: The project name in which the table is located
           c.)table_name: Name of the table.
       Result: You will get a three column matrix with the values distinct values(null not
        included),null_count and total count.
    """
    query = ("""With table as(
            SELECT COUNT(DISTINCT {col_name}) FROM {project_name}.{table_name})
            Select * from table ;
        """).format(col_name=column_name, project_name=project_name, table_name=table_name)
    distinct = main_func(query)
    distinct = distinct.rename(columns={'f0_': 'distinct'})
    query = """With table as(
            SELECT Count(*) as Count,{col_name} as {col_name}
            FROM (SELECT {col_name}  FROM {project_name}.{table_name})
            Group by {col_name}
            ORDER by COUNT(*) DESC)
            Select Count from table WHERE {col_name} is NULL;""".format(
                col_name=column_name, project_name=project_name, table_name=table_name)
    count_null = main_func(query)
    count_null = count_null.rename(columns={'Count': 'count_null'})
    query = ("""With table as(
            SELECT Count({col_name}) FROM {project_name}.{table_name})
            Select * from table ;
        """).format(col_name=column_name, project_name=project_name, table_name=table_name)
    total_count = main_func(query)
    total_count = total_count.rename(columns={'f0_': 'total_count'})
    result = pd.concat([distinct, count_null, total_count], axis=1)
    return result
def count_coverage_categorical(column_name, project_name, table_name, terms=10):
    """This function calculates count and coverage for a categorical column.
    Parameters to be passed:
        a.)column_name: Name of the categorical column for which
                        the count and coverage is to be calculated.
        b.)terms:number of terms in the categorical columns for
                which you want to calculate the count and coverage.
        c.)project_name: The project name in which the table is located
        d.)table_name: Name of the table.
    Result: A 3-columnar dataframe with the terms(arranged in descending
            order on the value of counts),count and coverage.
    """
```

```
    #This Query will be passed to the main function
    # for calculation of count and coverage after formatting.
    query = ("""With table as(
            SELECT Count(*) as Count,{col_name} as {col_name}
            FROM (SELECT {col_name}  FROM {project_name}.{table_name})
            Group by {col_name}
            ORDER by COUNT(*) DESC)
            Select {col_name},Count,Count*100/(Select Sum(COUNT) from table)as Coverage from table ORDER By
 Coverage DESC LIMIT {terms};
                    """).format(col_name=column_name,
                                terms=str(terms), project_name=project_name, table_name=table_name)
    result = main_func(query)
    #returning the result obtained.
    return result
def count_coverage_numeric(column_name, project_name, table_name, buckets=10):
    """This function will take the column_name and buckets as the input and will give an output with
    count,coverage for the numeric column.
    Parameters passed:
        a.)column_name:Name of the numeric column
        b.)buckets: Number of buckets to be made
        c.)project_name: The project name in which the table is located
        d.)table_name: Name of the table.
    Result: A 3-columnar dataframe with buckets,count and their coverage. """
    #This is the final query
    # that will be passed to the main function
    # after formatting the values of Query and column_name.
    query = dynamic_bucket(column_name, project_name, table_name, buckets)
    query_final = """With table as(
        SELECT Count(*) as Count,
        CASE
        {Query}
        END AS Buckets
        FROM (SELECT SAFE_CAST({col_name} AS FLOAT64) as {col_name} FROM {project_name}.{table_name})
        Group by Buckets
        ORDER by COUNT(*) DESC)
        Select Buckets,Count,Count*100/(Select Sum(COUNT) from table)as
        Coverage from table ORDER By Coverage DESC;""".format(
            Query=query, col_name=column_name, project_name=project_name, table_name=table_name)
    #calling the main function.
    result = main_func(query_final)
    #Note:in the result,there may be lesser buckets sometimes,
    # because in some buckets the count was 0.
```

```
    return result
def compare_leads_numeric(column_name, project_name, table_name, buckets=10):
    """This function will provide the comparison between converted and non-converted coverage
    for various buckets.
    Parameters passed:
        a.)column_name:Name of the numeric column
        b.)buckets: Number of buckets to be made
        c.)table_location: The address where the table is located on which
                        the user wants to perform some action.
    Result: A 3-columnar dataframe with buckets,converted coverage and non-converted coverage.
    Note:An inner join will be performed between the tables of converted coverage
        and non-converted coverage to find the common buckets. """
    query = dynamic_bucket(column_name, project_name, table_name, buckets)
    #This is the final query that will be passed to the main function
    # after formatting the values of Query and column_name.
    query_final = """With table as(
        SELECT Count(*) as Count,
        CASE
        {Query}
        END AS Buckets
label=0)FROM (SELECT SAFE_CAST({col_name} AS FLOAT64)as {col_name} FROM {project_name}.{table_name} where
        Group by Buckets),
        table_2 as(
        SELECT Count(*) as Count,
        CASE
        {Query}
        END AS Buckets_1
label=1)FROM (SELECT SAFE_CAST({col_name} AS FLOAT64)as {col_name} FROM {project_name}.{table_name} where
        Group by Buckets_1)
        Select a.Count*100/(Select Sum(COUNT) from table) as
non_converted_coverage,a.Buckets,b.Count*100/(Select Sum(COUNT) from table_2) as converted_coverage
        from table as a INNER JOIN table_2 as b
        ON a.Buckets=b.Buckets_1
        Order By a.Count DESC;""".format(Query=query,
                                         col_name=column_name,
                                         project_name=project_name,
                                         table_name=table_name)
    # calling the main function.
    result = main_func(query_final)
    #returning the result obtained.
    return result
```

```
def compare_leads_categorical(column_name, project_name, table_name, terms=10):

    """This function compares the leads values for a categorical column.
    Two tables are created one in which the data is sorted according to coverage_converted and
    the other is created on coverage_non_converted.
    Parameters:
        a.)column_name: Name of the numeric column for which you want
            to compare the converted and non-converted values.
        b.)terms: Number of categorical values for which you want to see the comparision.
        c.)project_name: The project name in which the table is located
        d.)table_name: Name of the table.
    Result: A 3-columnar data frame with converted coverage,terms ,non-converted coverage
    """
    query = ("""With table as(
            SELECT Count(*) as Count,{col_name} as {col_name}
            FROM (SELECT {col_name}  FROM {project_name}.{table_name} Where label=0)
            Group by {col_name}
            ORDER by COUNT(*) DESC),
            table_2 as(
            SELECT Count(*) as Count,{col_name} as {col_name}
            FROM (SELECT {col_name}  FROM {project_name}.{table_name} Where label=1)
            Group by {col_name}
            ORDER by COUNT(*))
            (Select a.Count*100/(Select Sum(COUNT) from table) as
non_converted_coverage,a.{col_name},b.Count*100/(Select Sum(COUNT) from table_2) as converted_coverage
            from table as a INNER JOIN table_2 as b
            ON a.{col_name}=b.{col_name}
            Order BY converted_coverage DESC
            LIMIT {terms})
            UNION DISTINCT
            (Select a.Count*100/(Select Sum(COUNT) from table) as
non_converted_coverage,a.{col_name},b.Count*100/(Select Sum(COUNT) from table_2) as converted_coverage
            from table as a INNER JOIN table_2 as b
            ON a.{col_name}=b.{col_name}
            Order BY non_converted_coverage DESC
            LIMIT {terms});""").format(col_name=column_name,
                                       terms=str(terms),
                                       project_name=project_name,
                                       table_name=table_name)
    result = main_func(query)
    return result
def main_func(query_passed):
```

```
    """ This function takes the query as the parameter and runs
    it on bigquery to generate a table which is
    then converted to dataframe.

    Parameters:
        a.)query_passed:The query which is to be run on bigquery.
    Result:The table generated by the query will be converted to a dataframe."""
    client = bigquery.Client()
    data = client.query(query_passed).to_dataframe()
    return data
def test_func(column_list, project_name, table_name, terms=10, buckets=10):
    """This function is created to test
    all the functions that were created
    for columns.
    Parameters required:a.)column_list:Name of all columns for which univariate
                            analysis is to be carried out.
                        b.)project_name: The project name in which the table is located
                        c.)table_name: Name of the table."""
    for i in column_list:
        column_name = i
        result = column_info(column_name, project_name, table_name)
        if (result.iloc[0, :] == "INT64").bool() or (result.iloc[0, :] == "FLOAT64").bool():
            bucket_limits = dynamic_bucket(column_name, project_name, table_name, buckets)
            print(bucket_limits)
            overview = numeric_data_overview(column_name, project_name, table_name, buckets)
            print(overview)
            count_coverage = count_coverage_numeric(column_name, project_name, table_name, buckets)
            print(count_coverage)
            compare_numeric = compare_leads_numeric(column_name, project_name, table_name, buckets)
            print(compare_numeric)
        if (result.iloc[0, :] == "STRING").bool():
            categorical_view = categorical_overview(column_name, project_name, table_name)
            print(categorical_view)
            count_coverage = count_coverage_categorical(column_name, project_name, table_name, terms)
            print(count_coverage)
            compare_categorical = compare_leads_categorical(
                column_name, project_name, table_name, terms)
            print(compare_categorical)
```

B.)Decision Tree

```
"""This module consists of functions that helps to generate
 a decision tree automatically after finding the
best hyperparameters using grid search cross validation
and outputs the precision-recall and accuracy score in the end.."""
import datetime
import pickle
from google.cloud import bigquery
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
def main_func(query_passed):
    """ This function takes the query as the parameter and runs
    it on bigquery to generate a table which is
    then converted to dataframe.
    Parameters:
        a.)query_passed:The query which is to be run on bigquery.
    Result:The table generated by the query will be converted to a dataframe."""
    client = bigquery.Client()
    data = client.query(query_passed).to_dataframe()
    return data
def count_coverage(column_name, project_name, table_name, threshold):
    """This function calculates count and coverage for a categorical column.
    Parameters to be passed:
        a.)column_name: Name of the column for which
                        the count and coverage is to be calculated.
        b.)project_name: The project name in
                        which the table is located
        c.)table_name: Name of the table.
        d.)threshold: If the amount of any particular value(in percentage) in
                a column , is greater than the threshold
                then that particular column will be not
                be considered while making the list for
                dataset generation.
```

```
        Result: A 3-columnar dataframe with the terms(arranged in descending
            order on the value of counts),count and coverage.

    """
    #This Query will be passed to the main function
    # for calculation of count and coverage after formatting.
    query = ("""With table as(
            SELECT Count(*) as Count,{col_name} as {col_name}
            FROM (SELECT {col_name}
            FROM {project_name}.{table_name})
            Group by {col_name}
            ORDER by COUNT(*) DESC),
            table_2 as(
            Select {col_name},Count,Count*100/(Select Sum(COUNT) from table)as Coverage
            from table
            ORDER By Coverage DESC )
            Select * from table_2 where Coverage>{threshold};
        """).format(col_name=column_name,
                    threshold=str(threshold), project_name=project_name, table_name=table_name)
    result = main_func(query)
    #returning the result obtained.
    return result
def main_dt_list(column_list, project_name, table_name, threshold):
    """This function generates the list of columns
    that are to be considered for dataset generation.
    Parameters required: a)column_list:All the names of the columns
                        that you want to consider for dataset generation.
                        b.)project_name: The project name in
                        which the table is located
                        c.)table_name: Name of the table.
                        d.)threshold: amount of null values in a column
                        that can be tolerated in percentage.
    Result: A list with names of columns that has coverage for none
            of its values greater than the threshold."""
    #the list for final dataset generation.
    final_columns = []
    for i in column_list:
        #passing the value to count_coverage function.
        data = count_coverage(i, project_name, table_name, threshold)
        #returned data frame contains values which has
        #coverage over the threshold limit.
        if data.empty:
```

```
        #column_name is added to the list.
            final_columns.append(i)
        elif i == "label":

            #considering label will be imbalanced
            #most of the time and we cannot drop that column
            #this condition allows the addition of label to
            #column_list even if it crosses the threshold.
            final_columns.append(i)
    return final_columns
def null_fill(column_list, project_name, table_name, threshold):
    """This function is used to administer the process of
    filling null values after findind whether a column
    is string or numeric and directs them to their particular
    null fill functions.
    Parameters required:a)column_list:All the names of the columns
                        that you want to consider for dataset generation.
                        b.)project_name: The project name in which the table is located
                        c.)table_name: Name of the table.
                        d.)threshold: amount of null values in a column that can be
                        tolerated in percentage.
    Result: A dataframe with all the columns asked along with additional time columns
            with no null values."""
    col_list = main_dt_list(column_list, project_name, table_name, threshold)
    # creation of an empty dataframe with all
    #the final column list values.
    result = pd.DataFrame(columns=col_list)
    for i in col_list:
        #query to check the column datatype.
        query = """WITH column_numeric as(
                    SELECT DATA_TYPE,COLUMN_NAME FROM {project_name}.INFORMATION_SCHEMA.COLUMNS
                    WHERE table_name="{table_name}")
                    Select DATA_TYPE FROM column_numeric
                    WHERE COLUMN_NAME='{col_name}'""".format(
                        project_name=project_name, table_name=table_name, col_name=i)
        data = main_func(query)
        #checks whether the datatype is string or Integer or float.
        if (data.iloc[0, 0] == "INT64") or (data.iloc[0, 0] == "FLOAT64"):
            data = numeric_na_fill(i, project_name, table_name)
        elif (data.iloc[0, 0] == "STRING"):
            data = categorical_na_fill(i, project_name, table_name)
        #inserts the data after filling of null values
```

```
        #in the same column_value in the empty dataframe
        #that we formed earlier in this function.a
        result[i] = data[i]
    # passes to time_data function for breaking down utc format

    # to hours,months,day of week, week of year.
    final_result = time_data(result, project_name, table_name)
    return final_result

def time_data(data, project_name, table_name):
    """This function checks the presence of visitStartTime column
    and if it is present, it divides the column into hour, week_day,
    week_year and day_month, else just returns the passed value if the
    column is not found.
    Parameters required:a.)data: data formed after filling of null
                            values.
                        b.)project_name: The project name in which
                        the table is located
                        c.)table_name: Name of the table.
    Result: Data with added time columns if visitStartTime was there in the passed dataset."""
    #checks whether the visitStartTime column is present in the dataset.
    if 'visitStartTime' in data.columns:
        query = """SELECT
                    FORMAT_TIMESTAMP('%A',dt_format) AS week_day,
                    FORMAT_TIMESTAMP('%H',dt_format) AS hour,
                    FORMAT_TIMESTAMP('%W',dt_format) AS week_year,
                    FORMAT_TIMESTAMP('%e',dt_format) AS day_month
                    FROM (
                        SELECT
                            TIMESTAMP_SECONDS(visitStartTime) AS dt_format
                        FROM
                            {project_name}.{table_name})
                    """.format(project_name=project_name, table_name=table_name)
        data_dt = main_func(query)
        data_dt[["hour", "week_year", "day_month"]] = data_dt[["hour", "week_year", "day_month"]].apply(
            pd.to_numeric)
        #since the new generated columns may have null values
        #as they were formed after the null_value filling process
        #so these steps are used to fill the null values .
        #the new generated columns are combined with the
        #passed dataset.
        data = data.reset_index(drop=True)
```

```
        data_dt = data_dt.reset_index(drop=True)
        final_data = pd.concat([data, data_dt], axis=1)
        #to avoid redundancy we drop the visitStartTime column.
        final_data = final_data.drop(["visitStartTime"], axis=1)
        data = final_data

    return data
def null_coverage(column_name, project_name, table_name):
    """This function calculates the percentage of null values
    that are present in a numeric or categorical column
    Parameters required:a.)column_name:Name of the column for
                            which the null_coverage is to be
                            counted.
                        b.)project_name: The project name in which
                        the table is located
                        c.)table_name: Name of the table.
    Result: A 1x1 dataframe with null coverage value."""
    query = ("""With table as(
            SELECT Count(*) as Count,{col_name} as {col_name}
            FROM (SELECT {col_name}  FROM {project_name}.{table_name})
            Group by {col_name}
            ORDER by COUNT(*) DESC),
            table_2 as(
            Select {col_name},Count,Count*100/(Select Sum(COUNT) from table)as Coverage
            from table ORDER By Coverage DESC)
            Select Coverage from table_2 where {col_name} IS NULL;
        """).format(col_name=column_name, project_name=project_name, table_name=table_name)
    result = main_func(query)
    #returning the result obtained.
    return result
def numeric_na_fill(column_name, project_name, table_name):
    """ This function checks the percentage of null in numeric column calculated using
    null coverage and then if the threshold is less than 10, fills the null values with median
    and if it is more than 10 then fills with an extreme value.
    Parameters required:a.)column_name:Name of the column for
                            which the null_coverage is to be
                            counted.
                        b.)project_name: The project name in which the table is located
                        c.)table_name: Name of the table.
    Result: A singular column dataframe with the column data generated
            after filling the null values for the specified column name."""
    query = """Select {col_name} from {project_name}.{table_name}""".format(
```

```
        col_name=column_name, project_name=project_name, table_name=table_name
    )
    col_data = main_func(query)
    data = null_coverage(column_name, project_name, table_name)
    #in case of no null values.
    if data.empty:

        query_final = """Select {col_name} from {project_name}.{table_name}""".format(
            col_name=column_name, project_name=project_name, table_name=table_name)
        result = main_func(query_final)
    #null value coverage less than 10,then fill with median.
    else:
        if data['Coverage'].iloc[0] <= 10:

            query_final = """Select IFNULL({col_name},{value}) {col_name}
            FROM (SELECT SAFE_CAST({col_name} AS FLOAT64) as {col_name}
            FROM {project_name}.{table_name})""".format(
                col_name=column_name, value=str(col_data.median().values[0]),
                project_name=project_name, table_name=table_name
            )
            result = main_func(query_final)
        #nul value coverage less than 10,then fill with extreme value.
        if data['Coverage'].iloc[0] > 10:
            query_final = """Select IFNULL({col_name},{value}) {col_name}
            FROM (SELECT SAFE_CAST({col_name} AS FLOAT64) as {col_name}
            FROM {project_name}.{table_name})""".format(
                col_name=column_name, value=str(-9999999999),
                project_name=project_name, table_name=table_name)
            result = main_func(query_final)
    return result
def categorical_na_fill(column_name, project_name, table_name):
    """ This function checks the percentage of null in categorical column calculated using
    null coverage and then if the threshold is less than 10, fills the null values with mode
    and if it is more than 10 then fills with not set.
    Parameters required:a.)column_name:Name of the column for
                            which the null_coverage is to be
                            counted.
                        b.)project_name: The project name in which the table is located
                        c.)table_name: Name of the table.
    Result: A singular column dataframe with the column data generated
            after filling the null values for the specified column name."""
    query = """Select {col_name} from {project_name}.{table_name}""".format(
```

```
        col_name=column_name, project_name=project_name, table_name=table_name
    )
    col_data = main_func(query)
    data = null_coverage(column_name, project_name, table_name)
    #in case of no null value.
    if data.empty:
        query_final = """Select {col_name} from {project_name}.{table_name}""".format(

            col_name=column_name, project_name=project_name, table_name=table_name)
        result = main_func(query_final)
    #in case of null value of less than 10% coverage.
    else:
        if data['Coverage'].iloc[0].values[0] <= 10:
            query_final = """Select IFNULL({col_name},"{value}") {col_name}
            FROM (SELECT {col_name} FROM {project_name}.{table_name})""".format(

                col_name=column_name, value=str(col_data.mode().values[0]),
                project_name=project_name, table_name=table_name)
            result = main_func(query_final)
        #in case of null value of more than 10% coverage.
        if data['Coverage'].iloc[0].values[0] > 10:
            query_final = """Select IFNULL({col_name},"{value}") {col_name}
            FROM (SELECT {col_name} FROM {project_name}.{table_name})""".format(
                col_name=column_name, value="not set",
                project_name=project_name, table_name=table_name)
            result = main_func(query_final)
    return result
def grouping(columns, project_name, table_name, threshold, cat_threshold):
    """This function takes all the categorical columns and checks
    whether the number of unique values in that column are more than
    a certain threshold,if it is more,then it categorizes all the values
    above the threshold in 'Others' .
    Parameters required:a)column_list:All the names of the columns
                        that you want to consider for dataset generation.
                        b.)project_name: The project name in which the table is located
                        c.)table_name: Name of the table.
                        d.)cat_threshold: The number of values after which every value
                        will be considered under 'Others
    Result: A multi columnar data frame with all categorical columns with max (threshold+1)
            unique values."""
    #formation of dataset with null value filling.
    answer = null_fill(columns, project_name, table_name, threshold)
```

```
    for i in columns:
        query = """WITH column_numeric as(
                    SELECT DATA_TYPE,COLUMN_NAME FROM {project_name}.INFORMATION_SCHEMA.COLUMNS
                    WHERE table_name="{table_name}")
                    Select DATA_TYPE FROM column_numeric
                    WHERE COLUMN_NAME='{col_name}'""".format(
                        project_name=project_name, table_name=table_name, col_name=i)
        data = main_func(query)

        #checks whether any passed column is of categorical type.
        if (data.iloc[0, 0] == "STRING"):
            #checks whether the number of unique columns in a
            # categorical column is more than the threshold.
            if answer[i].nunique() > cat_threshold:
                #takes the count of for the threshold value.
                count =
answer[i].value_counts()[0:cat_threshold].to_frame().reset_index(drop=True).iloc[-1, :].values[0]

                #all the counts for different values of the column.
                column_count = answer[i].value_counts()
                #column values whose count is lesser than the threshold value count.
                column_values = answer[i].isin(column_count.index[column_count < count])
                #replaces the values whose count is lesser than a certain limit.
                answer.loc[column_values, i] = "Others"
    #one-hot encoding of the dataset.
    result = pd.get_dummies(answer)
    return result
def decision_tree(column_list, project_name, table_name, threshold=80, cat_threshold=10):
    """This function helps in generation of the decision tree
    automatically and provides the user with values like precision,
    accuracy,recall,f1-score.
    Parameters required:a)column_list:All the names of the columns
                        that you want to consider for dataset generation.
                        b.)project_name: The project name in which the table is located
                        c.)table_name: Name of the table.
                        d.)threshold: amount of null values in a column that can be
                        tolerated in percentage.
                        e.) cat_threshold: number of values above which any value
                        in the categorical column will be considered as "Others"
    Result:Accuracy rate,classification report and confusion matrix will be formed
            on the basis of the decision tree generated."""
    data = grouping(column_list, project_name, table_name, threshold, cat_threshold)
```

```
train = data.drop(['label'], axis=1)
train_label = data['label']
x_train, x_test, y_train, y_test = train_test_split(
    train, train_label, test_size=0.3, random_state=0)
param_dist = {
    'max_depth':[5, 20],
    'min_samples_leaf':[5, 20]
}
tree = DecisionTreeClassifier()

tree_cv = GridSearchCV(tree, param_dist, cv=2)
tree_cv.fit(x_train, y_train)
print('Tuned Decision Tree Parameters:{}'.format(tree_cv.best_params_))
print('Best Score:{}'.format(tree_cv.best_score_))
y_pred_class = tree_cv.predict(x_test)
accuracy = metrics.accuracy_score(y_test, y_pred_class)
print('Accuracy: {0:0.2f}'.format(
    accuracy))
print(confusion_matrix(y_test, y_pred_class))

classification_report_ = classification_report(y_test, y_pred_class, zero_division=1)
print(classification_report_)
filename = 'finalized_model.sav'
pickle.dump(tree_cv, open(filename, 'wb'))
```

CONCLUSION AND FUTURE WORK

The project has been complete with a 97.03% accuracy and 80% precision and recall. The project helped us formulate a decision tree and generate insights on the classification, which will also help the company to strategize the marketing campaigns for their next quarter.

Also,the front-end for the application was to be made and is under progress. The front end is to be made using flask with the decision tree model ,database and several other models running in the back-end. This front end application will help the user to provide the column names as the input and get the univariate analysis along with insights in the output in a visual form which will help him in making plans.

REFERENCES

1.)https://console.cloud.google.com/bigquery?project=project-chatbot-cmnfwk&redirect_from_classic=true

2.)https://stackoverflow.com/questions/

3.)https://cloud.google.com/bigquery/docs/reference/libraries

4.)https://googleapis.dev/python/bigquery/latest/index.html

5.)https://console.cloud.google.com/compute/instances?project=tatvic-gcp-dev-team&folder&organizationId=1010474793150&instancessize=50

6.)https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/

7.)https://www.dezyre.com/recipes/optimize-hyper-parameters-of-decisiontree-model-using-grid-search-in-python

8.)https://towardsdatascience.com/exploring-univariate-data-e7e2dc8fde80

9.)https://towardsdatascience.com/build-the-story-around-data-using-exploratory-data-analysis-and-pandas-c85bf3beff87