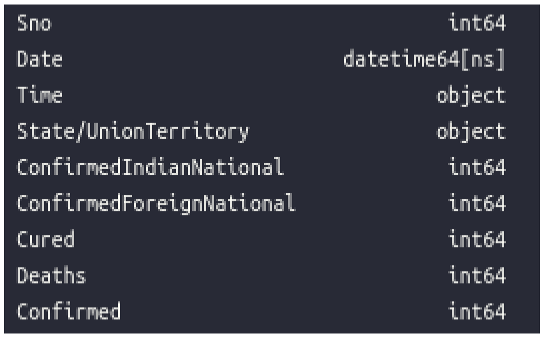
ML Assignment-1 Report | Divyansh Rastogi (2019464)

Q1.

Dataset: covid_19_india.csv

Approach:

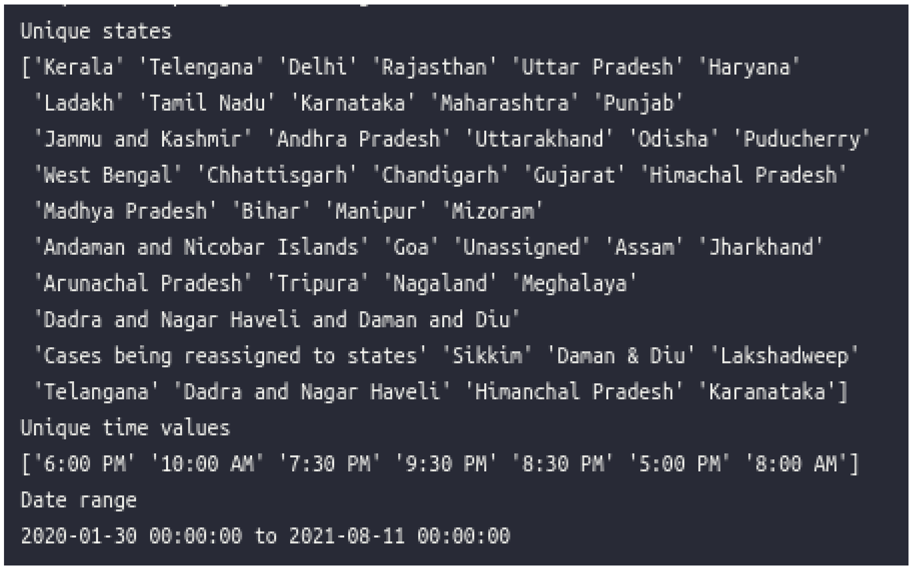
1. **Preprocessing:**
    a. Date column is converted to pandas datetime series.
    b. ConfirmedIndianNational and ConfirmedForeignNational contain '-' as NaN values, where it is replaced with a 0 for the sole purpose of converting these columns to numeric values.
    c. Some states in State/UnionTerritory, have a '*' in their name, which is also removed in preprocessing.
2. **[1.1]** Modified dtypes and columns of dataset.



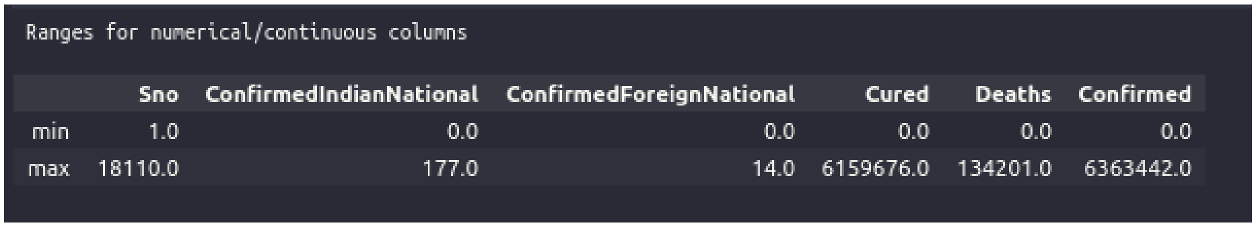
```
Sno                                 int64
Date                       datetime64[ns]
Time                               object
State/UnionTerritory               object
ConfirmedIndianNational             int64
ConfirmedForeignNational            int64
Cured                               int64
Deaths                              int64
Confirmed                           int64
```

   Time & State/UnionTerritory columns are categorical while all others are continuous.

3. **[1.1]** Possible data values and ranges



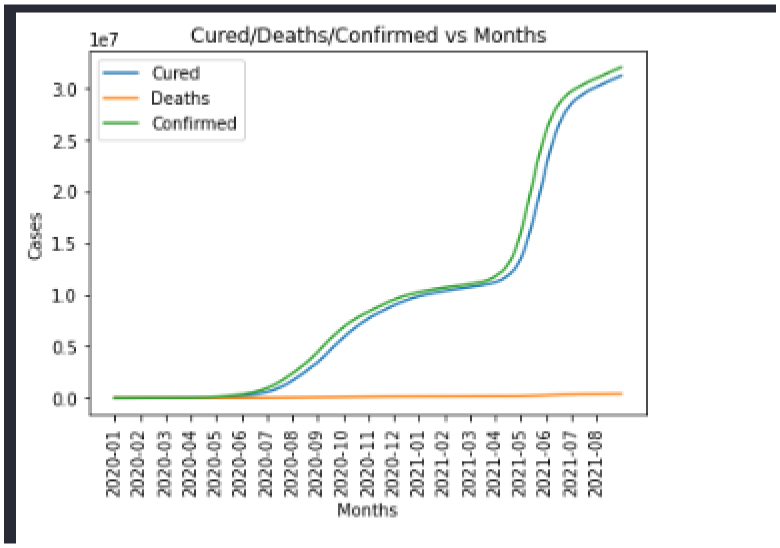
```
Unique states
['Kerala' 'Telengana' 'Delhi' 'Rajasthan' 'Uttar Pradesh' 'Haryana'
 'Ladakh' 'Tamil Nadu' 'Karnataka' 'Maharashtra' 'Punjab'
 'Jammu and Kashmir' 'Andhra Pradesh' 'Uttarakhand' 'Odisha' 'Puducherry'
 'West Bengal' 'Chhattisgarh' 'Chandigarh' 'Gujarat' 'Himachal Pradesh'
 'Madhya Pradesh' 'Bihar' 'Manipur' 'Mizoram'
 'Andaman and Nicobar Islands' 'Goa' 'Unassigned' 'Assam' 'Jharkhand'
 'Arunachal Pradesh' 'Tripura' 'Nagaland' 'Meghalaya'
 'Dadra and Nagar Haveli and Daman and Diu'
 'Cases being reassigned to states' 'Sikkim' 'Daman & Diu' 'Lakshadweep'
 'Telangana' 'Dadra and Nagar Haveli' 'Himanchal Pradesh' 'Karanataka']
Unique time values
['6:00 PM' '10:00 AM' '7:30 PM' '9:30 PM' '8:30 PM' '5:00 PM' '8:00 AM']
Date range
2020-01-30 00:00:00 to 2021-08-11 00:00:00
```



```
Ranges for numerical/continuous columns

        Sno  ConfirmedIndianNational  ConfirmedForeignNational      Cured    Deaths  Confirmed
min     1.0                      0.0                       0.0        0.0       0.0        0.0
max 18110.0                    177.0                      14.0  6159676.0  134201.0  6363442.0
```

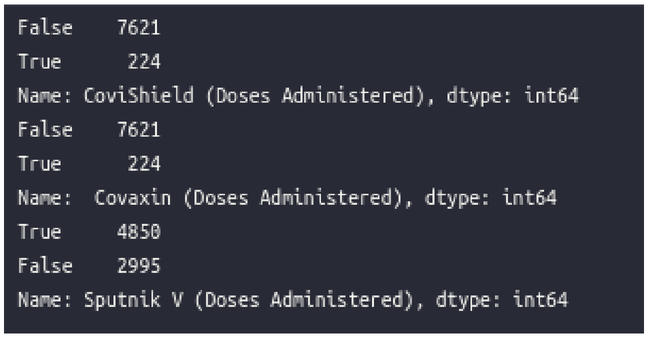
4. **[1.2]** For showing a timewise trend, [cured, deaths, confirmed] are first sum-grouped on a day to day basis. Although the trend is reflected in gaps of months to avoid clutter.



Dataset: covid_vaccine_statewise.csv

Approach:

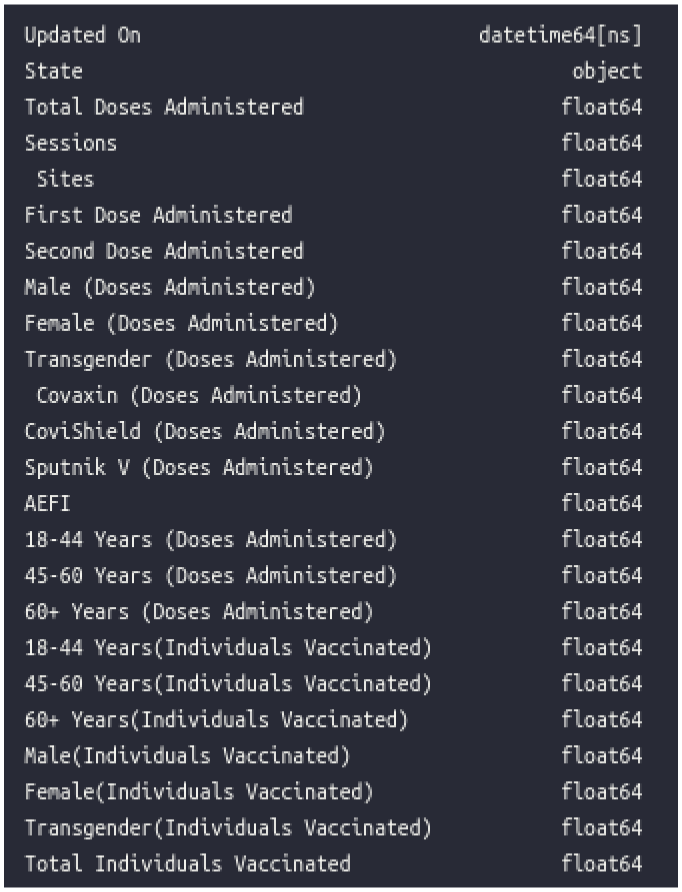
1. **Preprocessing**:
   a. 'Updated On' column is converted to pandas datetime column.
   b. Many columns contain NaN values here, but given the columns for data visualization, only NaN values for those columns are handled.
   c. NaN values are seen in Doses for Covaxin, Covishield and Sputnik. The concentration of such NaN values is shown below:



```
False    7621
True      224
Name: CoviShield (Doses Administered), dtype: int64
False    7621
True      224
Name:  Covaxin (Doses Administered), dtype: int64
True     4850
False    2995
Name: Sputnik V (Doses Administered), dtype: int64
```

As concentration of NaN values for Covishield and Covaxin is low, the rows pertaining to these NaN values are dropped. While for Sputnik V, as concentration is relatively high, all NaN values are replaced with 0. (As from general knowledge, it could be possible that values aren't recorded, as Sputnik V wasn't introduced to India in the earlier stages)

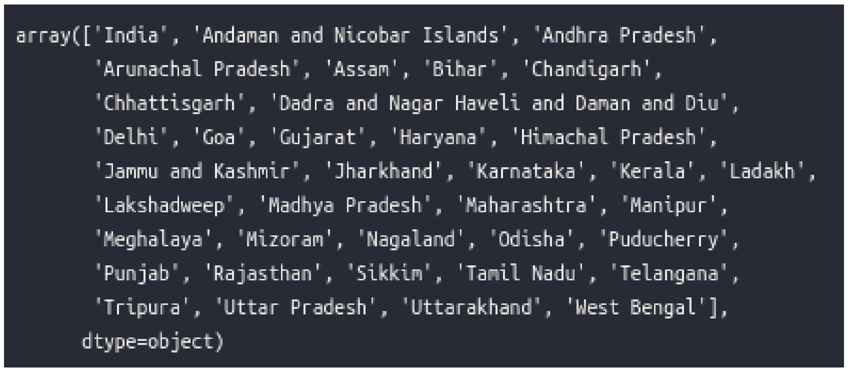
2. **[1.1]** Columns in the dataset with their dtypes



```
Updated On                                datetime64[ns]
State                                             object
Total Doses Administered                         float64
Sessions                                         float64
 Sites                                           float64
First Dose Administered                          float64
Second Dose Administered                         float64
Male (Doses Administered)                        float64
Female (Doses Administered)                      float64
Transgender (Doses Administered)                 float64
 Covaxin (Doses Administered)                    float64
CoviShield (Doses Administered)                  float64
Sputnik V (Doses Administered)                   float64
AEFI                                             float64
18-44 Years (Doses Administered)                 float64
45-60 Years (Doses Administered)                 float64
60+ Years (Doses Administered)                   float64
18-44 Years(Individuals Vaccinated)              float64
45-60 Years(Individuals Vaccinated)              float64
60+ Years(Individuals Vaccinated)                float64
Male(Individuals Vaccinated)                     float64
Female(Individuals Vaccinated)                   float64
Transgender(Individuals Vaccinated)              float64
Total Individuals Vaccinated                     float64
```

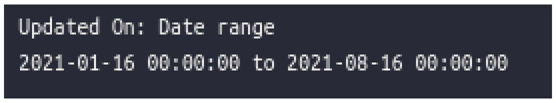
State is a categorical feature, all else are continuous features.

3. **[1.1]** Unique values for State:



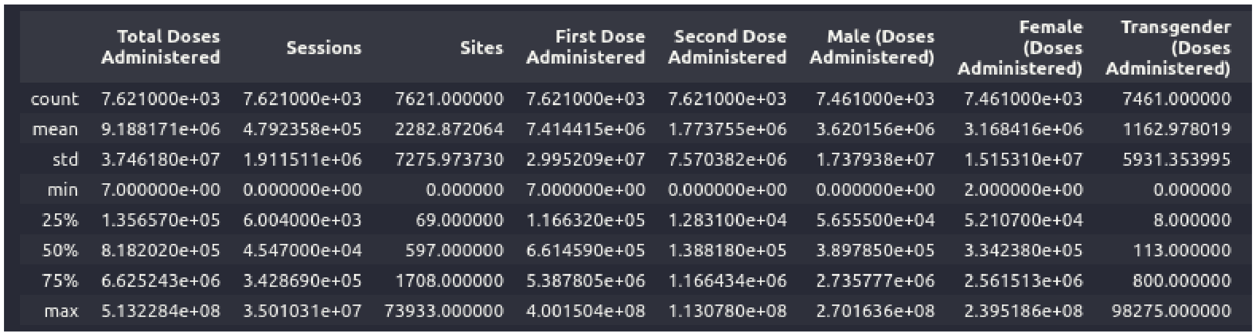
```
array(['India', 'Andaman and Nicobar Islands', 'Andhra Pradesh',
       'Arunachal Pradesh', 'Assam', 'Bihar', 'Chandigarh',
       'Chhattisgarh', 'Dadra and Nagar Haveli and Daman and Diu',
       'Delhi', 'Goa', 'Gujarat', 'Haryana', 'Himachal Pradesh',
       'Jammu and Kashmir', 'Jharkhand', 'Karnataka', 'Kerala', 'Ladakh',
       'Lakshadweep', 'Madhya Pradesh', 'Maharashtra', 'Manipur',
       'Meghalaya', 'Mizoram', 'Nagaland', 'Odisha', 'Puducherry',
       'Punjab', 'Rajasthan', 'Sikkim', 'Tamil Nadu', 'Telangana',
       'Tripura', 'Uttar Pradesh', 'Uttarakhand', 'West Bengal'],
      dtype=object)
```

4. **[1.1]** Ranges for continuous features:

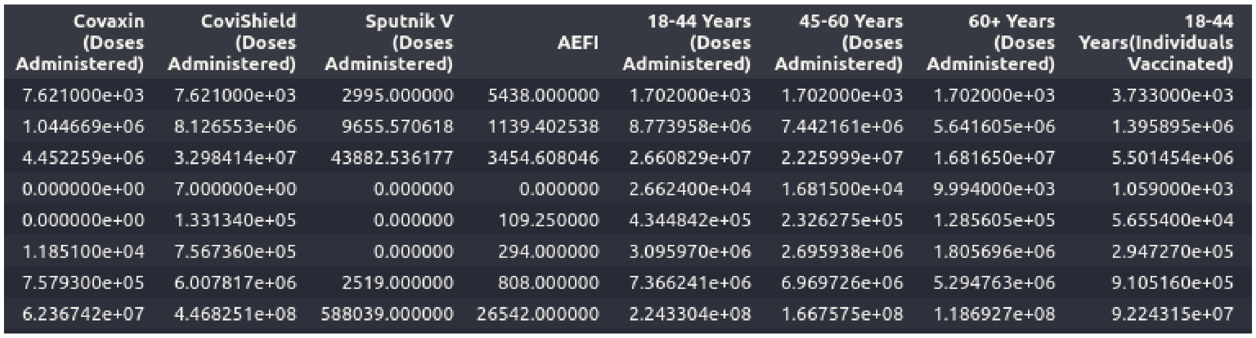


```
Updated On: Date range
2021-01-16 00:00:00 to 2021-08-16 00:00:00
```

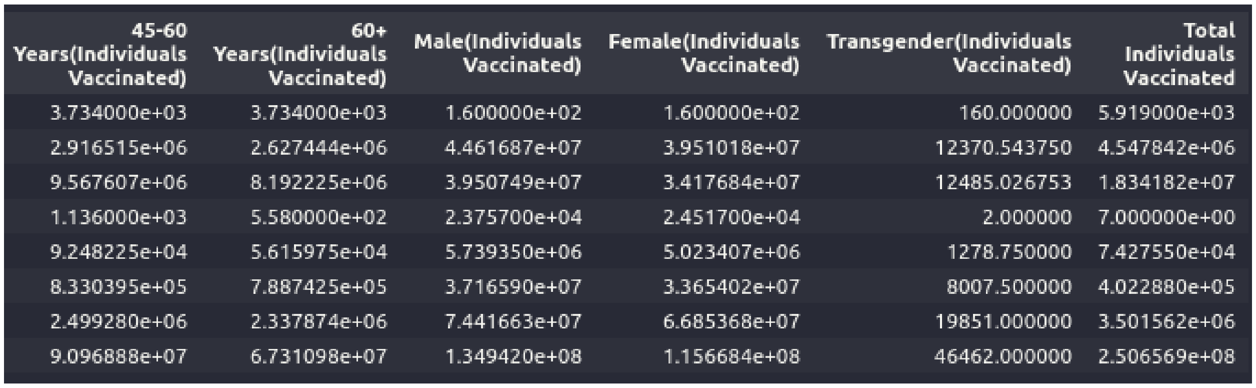
Using .describe() method. The 4th row (indexed min) & 8th row (indexed max) denote the ranges.



	Total Doses Administered	Sessions	Sites	First Dose Administered	Second Dose Administered	Male (Doses Administered)	Female (Doses Administered)	Transgender (Doses Administered)
count	7.621000e+03	7.621000e+03	7621.000000	7.621000e+03	7.621000e+03	7.461000e+03	7.461000e+03	7461.000000
mean	9.188171e+06	4.792358e+05	2282.872064	7.414415e+06	1.773755e+06	3.620156e+06	3.168416e+06	1162.978019
std	3.746180e+07	1.911511e+06	7275.973730	2.995209e+07	7.570382e+06	1.737938e+07	1.515310e+07	5931.353995
min	7.000000e+00	0.000000e+00	0.000000	7.000000e+00	0.000000e+00	0.000000e+00	2.000000e+00	0.000000
25%	1.356570e+05	6.004000e+03	69.000000	1.166320e+05	1.283100e+04	5.655500e+04	5.210700e+04	8.000000
50%	8.182020e+05	4.547000e+04	597.000000	6.614590e+05	1.388180e+05	3.897850e+05	3.342380e+05	113.000000
75%	6.625243e+06	3.428690e+05	1708.000000	5.387805e+06	1.166434e+06	2.735777e+06	2.561513e+06	800.000000
max	5.132284e+08	3.501031e+07	73933.000000	4.001504e+08	1.130780e+08	2.701636e+08	2.395186e+08	98275.000000

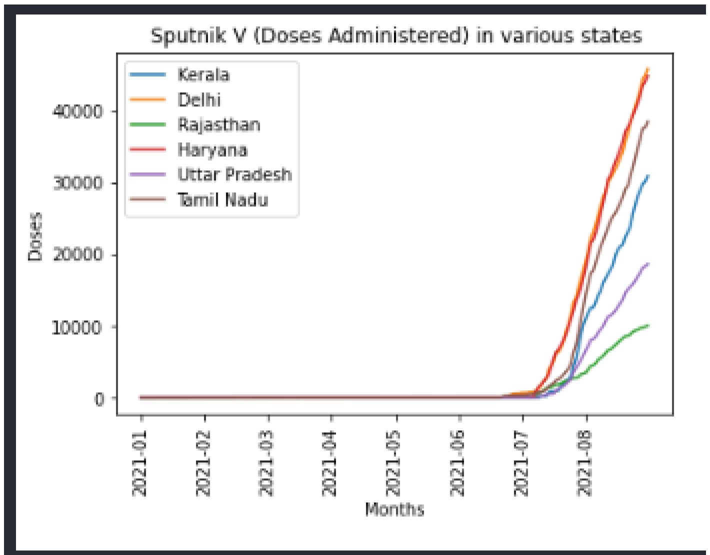


Covaxin (Doses Administered)	CoviShield (Doses Administered)	Sputnik V (Doses Administered)	AEFI	18-44 Years (Doses Administered)	45-60 Years (Doses Administered)	60+ Years (Doses Administered)	18-44 Years(Individuals Vaccinated)
7.621000e+03	7.621000e+03	2995.000000	5438.000000	1.702000e+03	1.702000e+03	1.702000e+03	3.733000e+03
1.044669e+06	8.126553e+06	9655.570618	1139.402538	8.773958e+06	7.442161e+06	5.641605e+06	1.395895e+06
4.452259e+06	3.298414e+07	43882.536177	3454.608046	2.660829e+07	2.225999e+07	1.681650e+07	5.501454e+06
0.000000e+00	7.000000e+00	0.000000	0.000000	2.662400e+04	1.681500e+04	9.994000e+03	1.059000e+03
0.000000e+00	1.331340e+05	0.000000	109.250000	4.344842e+05	2.326275e+05	1.285605e+05	5.655400e+04
1.185100e+04	7.567360e+05	0.000000	294.000000	3.095970e+06	2.695938e+06	1.805696e+06	2.947270e+05
7.579300e+05	6.007817e+06	2519.000000	808.000000	7.366241e+06	6.969726e+06	5.294763e+06	9.105160e+05
6.236742e+07	4.468251e+08	588039.000000	26542.000000	2.243304e+08	1.667575e+08	1.186927e+08	9.224315e+07

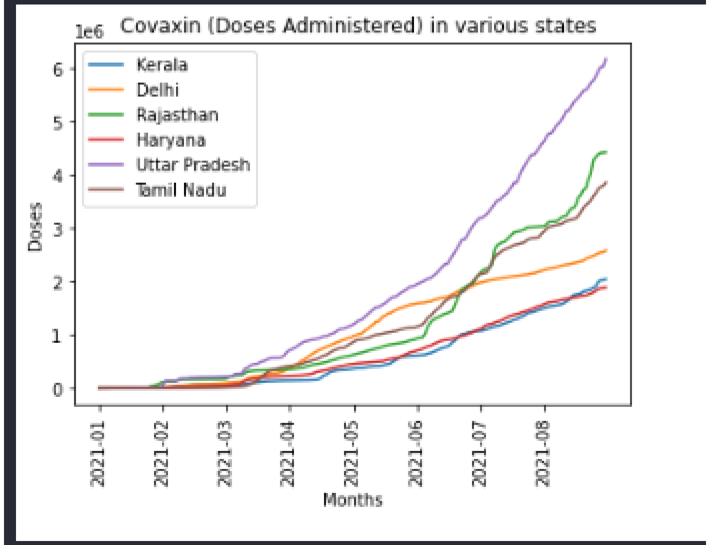


45-60 Years(Individuals Vaccinated)	60+ Years(Individuals Vaccinated)	Male(Individuals Vaccinated)	Female(Individuals Vaccinated)	Transgender(Individuals Vaccinated)	Total Individuals Vaccinated
3.734000e+03	3.734000e+03	1.600000e+02	1.600000e+02	160.000000	5.919000e+03
2.916515e+06	2.627444e+06	4.461687e+07	3.951018e+07	12370.543750	4.547842e+06
9.567607e+06	8.192225e+06	3.950749e+07	3.417684e+07	12485.026753	1.834182e+07
1.136000e+03	5.580000e+02	2.375700e+04	2.451700e+04	2.000000	7.000000e+00
9.248225e+04	5.615975e+04	5.739350e+06	5.023407e+06	1278.750000	7.427550e+04
8.330395e+05	7.887425e+05	3.716590e+07	3.365402e+07	8007.500000	4.022880e+05
2.499280e+06	2.337874e+06	7.441663e+07	6.685368e+07	19851.000000	3.501562e+06
9.096888e+07	6.731098e+07	1.349420e+08	1.156684e+08	46462.000000	2.506569e+08

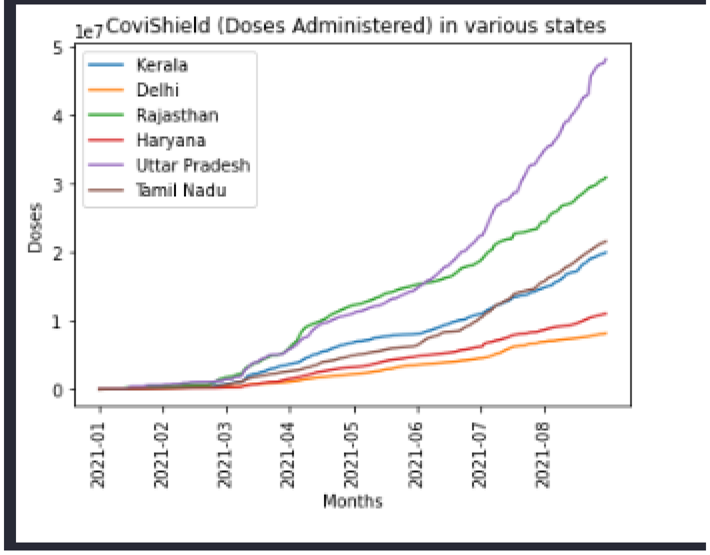
5. **[1.3]** Plotting is done vaccine-wise and state-wise.
    a. Vaccine-wise:



Sputnik V (Doses Administered) in various states
Kerala
Delhi
Rajasthan
Haryana
Uttar Pradesh
Tamil Nadu
Doses
0
10000
20000
30000
40000
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months

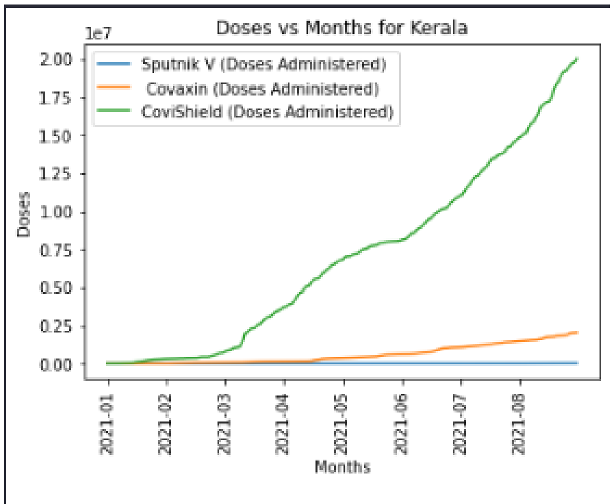


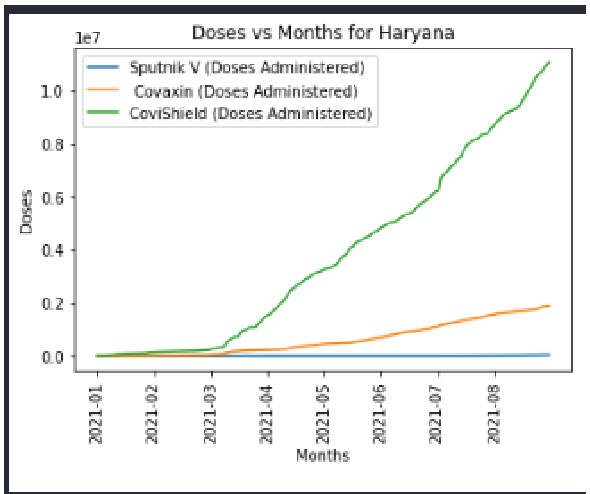
Covaxin (Doses Administered) in various states
1e6
Kerala
Delhi
Rajasthan
Haryana
Uttar Pradesh
Tamil Nadu
Doses
0
1
2
3
4
5
6
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months

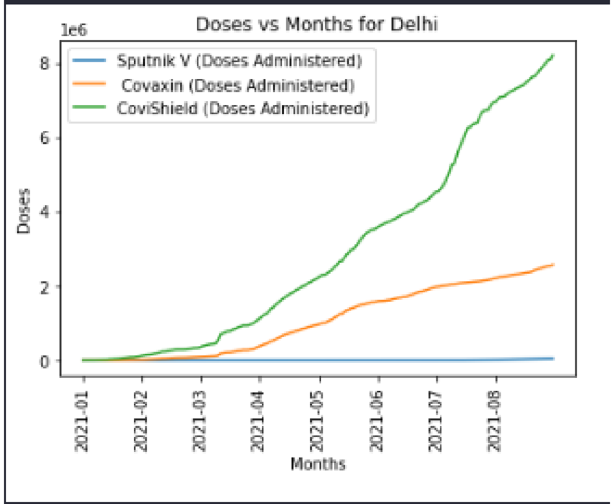


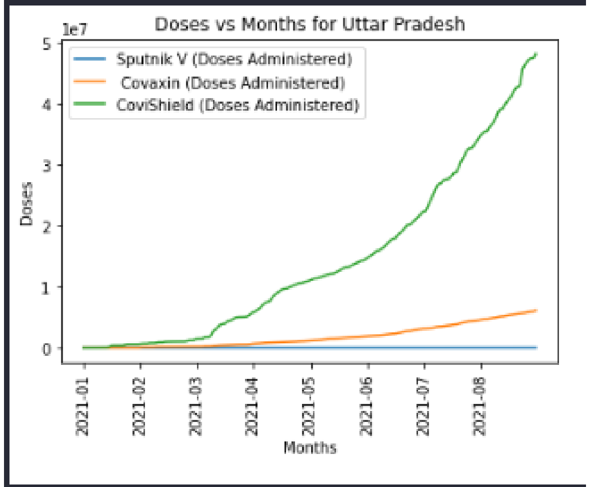
CoviShield (Doses Administered) in various states
1e7
Kerala
Delhi
Rajasthan
Haryana
Uttar Pradesh
Tamil Nadu
Doses
0
1
2
3
4
5
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months

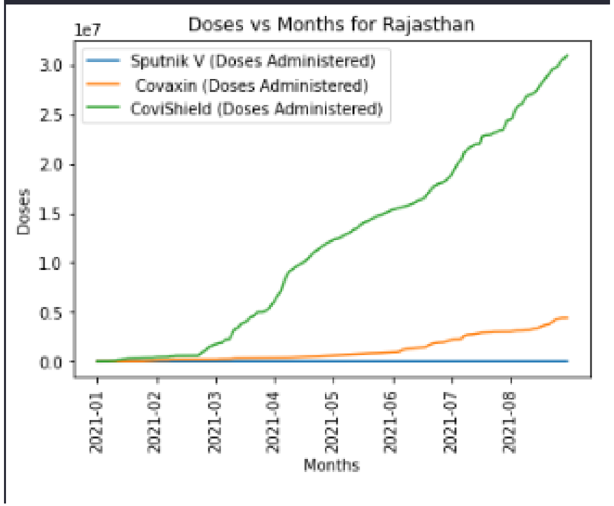
b. State-wise:

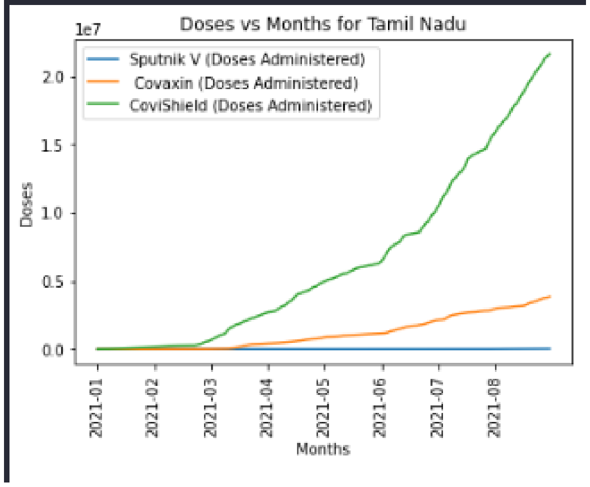

Doses vs Months for Kerala
1e7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Doses
Sputnik V (Doses Administered)
Covaxin (Doses Administered)
CoviShield (Doses Administered)
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months


Doses vs Months for Haryana
1e7
1.0
0.8
0.6
0.4
0.2
0.0
Doses
Sputnik V (Doses Administered)
Covaxin (Doses Administered)
CoviShield (Doses Administered)
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months


Doses vs Months for Delhi
1e6
8
6
4
2
0
Doses
Sputnik V (Doses Administered)
Covaxin (Doses Administered)
CoviShield (Doses Administered)
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months


Doses vs Months for Uttar Pradesh
1e7
5
4
3
2
1
0
Doses
Sputnik V (Doses Administered)
Covaxin (Doses Administered)
CoviShield (Doses Administered)
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months

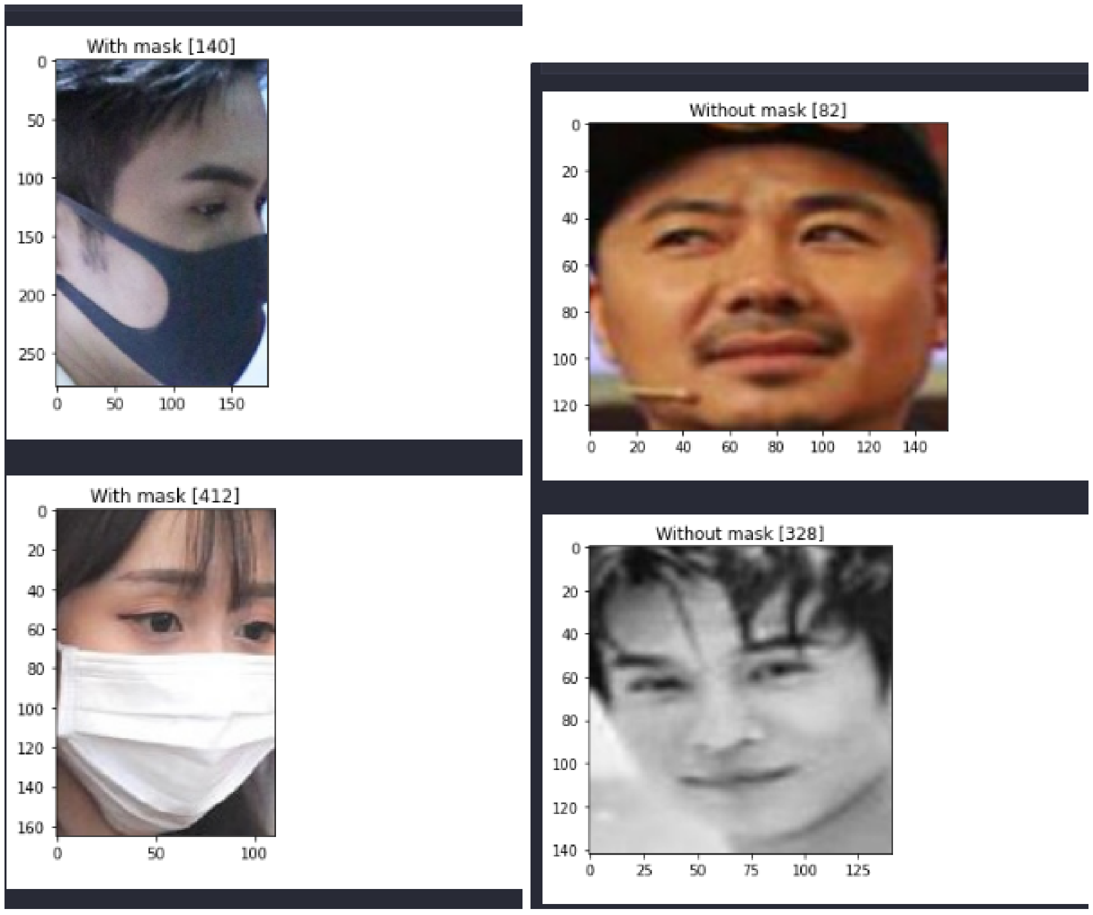

Doses vs Months for Rajasthan
1e7
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Doses
Sputnik V (Doses Administered)
Covaxin (Doses Administered)
CoviShield (Doses Administered)
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months


Doses vs Months for Tamil Nadu
1e7
2.0
1.5
1.0
0.5
0.0
Doses
Sputnik V (Doses Administered)
Covaxin (Doses Administered)
CoviShield (Doses Administered)
2021-01
2021-02
2021-03
2021-04
2021-05
2021-06
2021-07
2021-08
Months

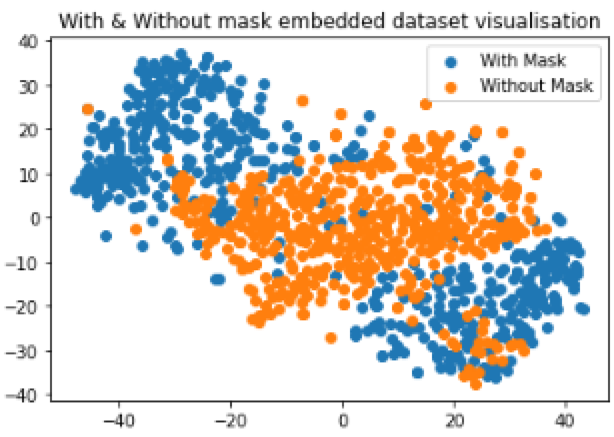
<u>Dataset:</u> Face mask dataset

<u>Approach:</u>

1. **[1.4.a]** 5 random images from each class (mask / no mask) are visualised using the imshow method of matplotlib.pyplot. Some examples:



2. **Preprocessing**:
    a. As image size is varying, all images are resized to (128, 128) using resize methods from pillow library.
3. **[1.4.b]** As t-sne is an iterative method, images are shuffled in both classes in our common input array and then flattened to pass to t-sne.
4. **[1.4.b]** Finally, using the method fit_transform of t-sne, the flattened 128x128 image vector is reduced to a 2d vector.
5. **[1.4.b]** The scatter plot for both the classes is observed as the following:

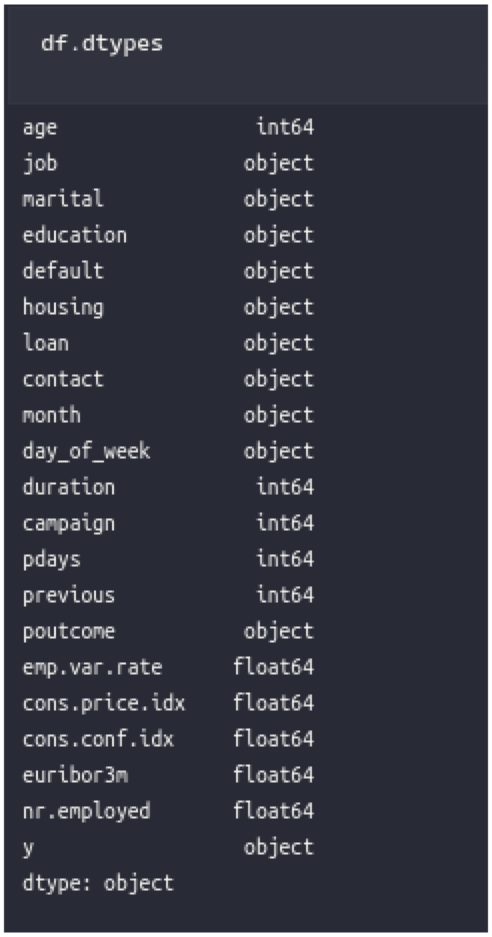


The classes are not linearly separable and slightly homogenized due to the presence of outliers which leads to almost no separability.
Disregarding the outliers, without-mask class could be seen as clustered to the center of with mask-class, which could lead to separation in higher dimensions such as a family of lines (3 dimensions) with first line separating cluster-1 (top left blue) and second line separating (bottom right blue) or separation in transformation of input space such as creating a radial boundary by using polar coordinates.

Q2.

Dataset: bank-additional-full.csv
Approach:
There is high class imbalance observed in data, almost 9:1.

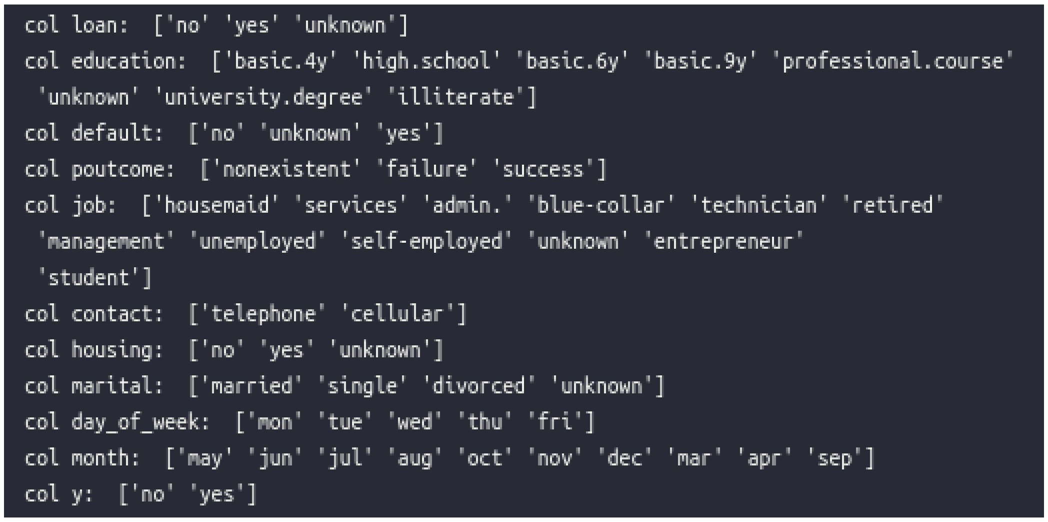
1. Analyse dataset through head() & describe() for the dataframe
2. Check .dtypes to analyse categorical columns



```
df.dtypes
```

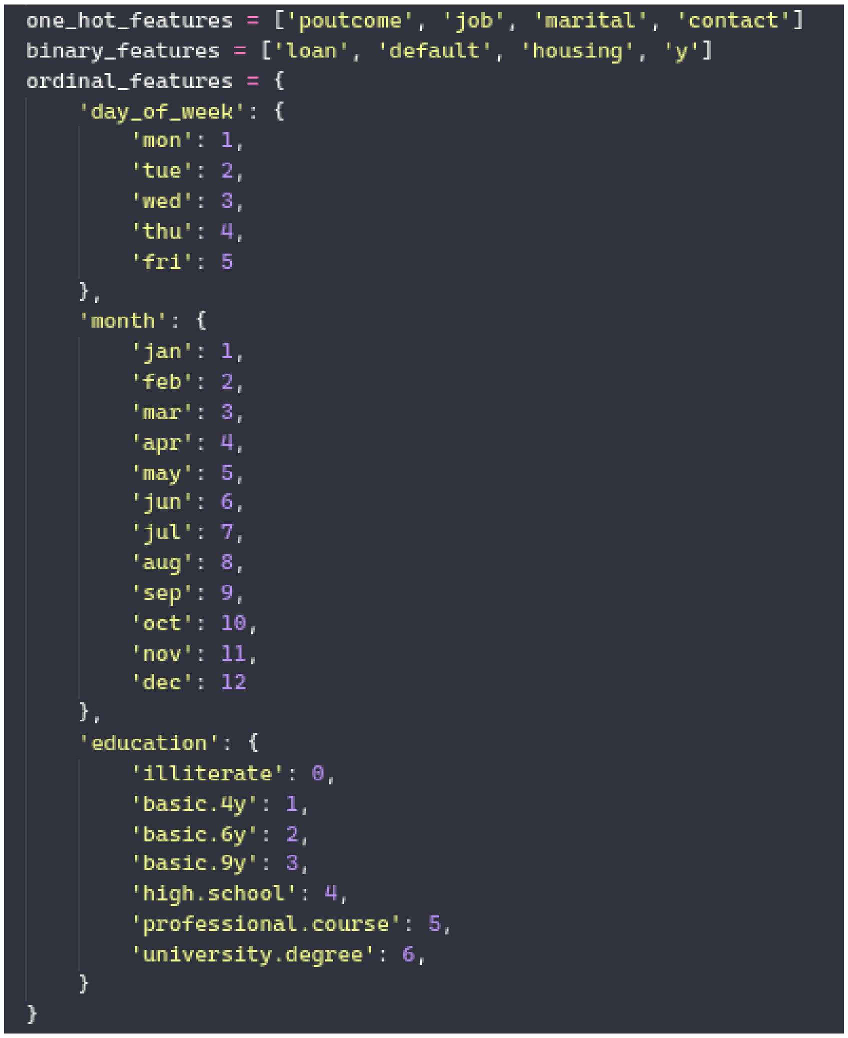
```
age                 int64
job                object
marital            object
education          object
default            object
housing            object
loan               object
contact            object
month              object
day_of_week        object
duration            int64
campaign            int64
pdays               int64
previous            int64
poutcome           object
emp.var.rate      float64
cons.price.idx    float64
cons.conf.idx     float64
euribor3m         float64
nr.employed       float64
y                  object
dtype: object
```

3. **Preprocessing**:
    a. No NaN values are recorded in the dataframe.
    b. Categorical features are segregated using ._get_numeric_data() on the dataframe.



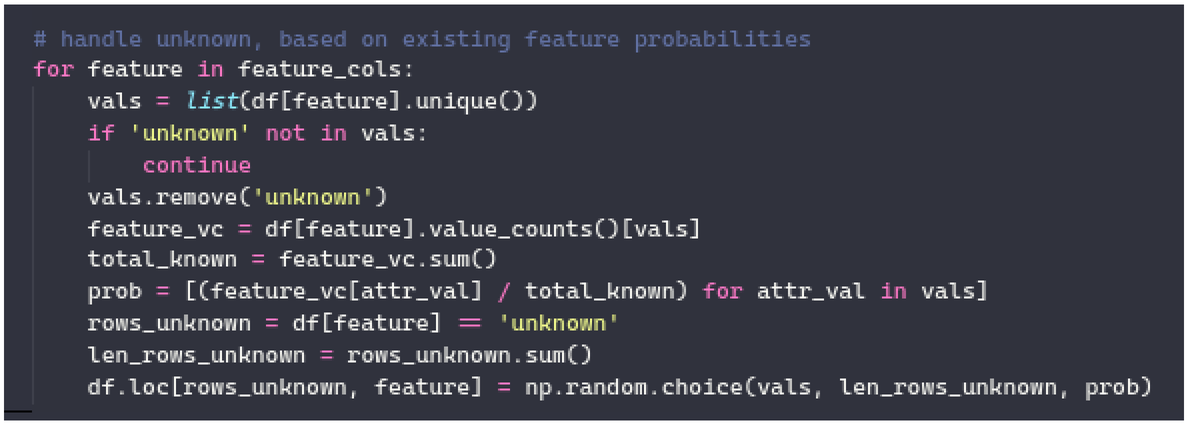
```
col loan:  ['no' 'yes' 'unknown']
col education:  ['basic.4y' 'high.school' 'basic.6y' 'basic.9y' 'professional.course'
 'unknown' 'university.degree' 'illiterate']
col default:  ['no' 'unknown' 'yes']
col poutcome:  ['nonexistent' 'failure' 'success']
col job:  ['housemaid' 'services' 'admin.' 'blue-collar' 'technician' 'retired'
 'management' 'unemployed' 'self-employed' 'unknown' 'entrepreneur'
 'student']
col contact:  ['telephone' 'cellular']
col housing:  ['no' 'yes' 'unknown']
col marital:  ['married' 'single' 'divorced' 'unknown']
col day_of_week:  ['mon' 'tue' 'wed' 'thu' 'fri']
col month:  ['may' 'jun' 'jul' 'aug' 'oct' 'nov' 'dec' 'mar' 'apr' 'sep']
col y:  ['no' 'yes']
```

c. For each categorical, unique values are listed. Based on these unique values, we divide the features into one_hot_features, binary_features & ordinal_features.



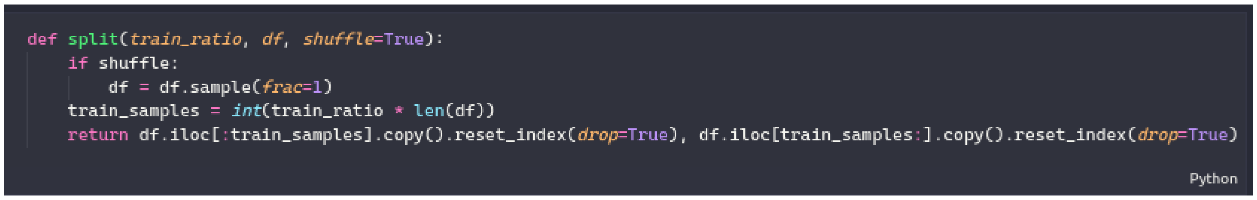
```
one_hot_features = ['poutcome', 'job', 'marital', 'contact']
binary_features = ['loan', 'default', 'housing', 'y']
ordinal_features = {
    'day_of_week': {
        'mon': 1,
        'tue': 2,
        'wed': 3,
        'thu': 4,
        'fri': 5
    },
    'month': {
        'jan': 1,
        'feb': 2,
        'mar': 3,
        'apr': 4,
        'may': 5,
        'jun': 6,
        'jul': 7,
        'aug': 8,
        'sep': 9,
        'oct': 10,
        'nov': 11,
        'dec': 12
    },
    'education': {
        'illiterate': 0,
        'basic.4y': 1,
        'basic.6y': 2,
        'basic.9y': 3,
        'high.school': 4,
        'professional.course': 5,
        'university.degree': 6,
    }
}
```

d. Binary features are yes/no features which are replaced with 1 and 0 resp.
e. Ordinal features are the features which have a sense of ordering to them and are replaced with the object as defined above.
f. One hot features are the features which have no relative ordering, thus are converted to one hot column.
g. Some categorical features have unknown values, thus for such columns, these unknown values are sampled from a weighted uniform distribution based on existing and known values of that feature.



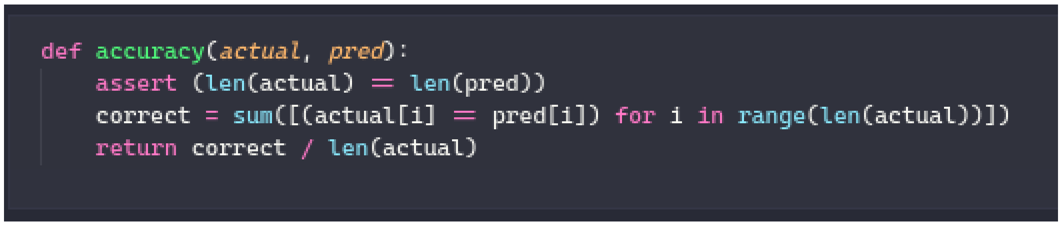
```python
# handle unknown, based on existing feature probabilities
for feature in feature_cols:
    vals = list(df[feature].unique())
    if 'unknown' not in vals:
        continue
    vals.remove('unknown')
    feature_vc = df[feature].value_counts()[vals]
    total_known = feature_vc.sum()
    prob = [(feature_vc[attr_val] / total_known) for attr_val in vals]
    rows_unknown = df[feature] == 'unknown'
    len_rows_unknown = rows_unknown.sum()
    df.loc[rows_unknown, feature] = np.random.choice(vals, len_rows_unknown, prob)
```

4. A custom function is used for splitting the data frame into training and testing sets. We first shuffle the rows of the dataframe and then split based on the ratio.



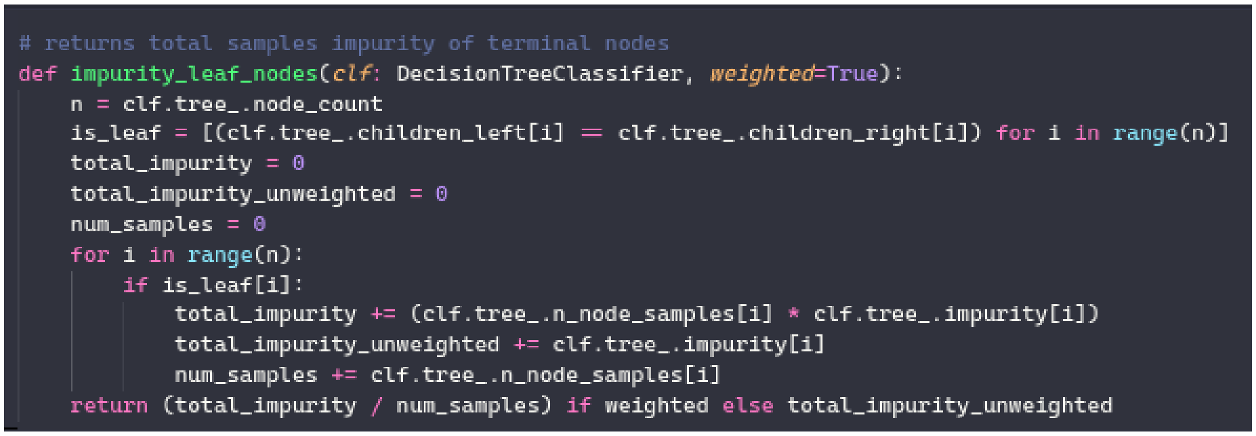
```python
def split(train_ratio, df, shuffle=True):
    if shuffle:
        df = df.sample(frac=1)
    train_samples = int(train_ratio * len(df))
    return df.iloc[:train_samples].copy().reset_index(drop=True), df.iloc[train_samples:].copy().reset_index(drop=True)
```

5. A custom function is used for calculating the accuracy.



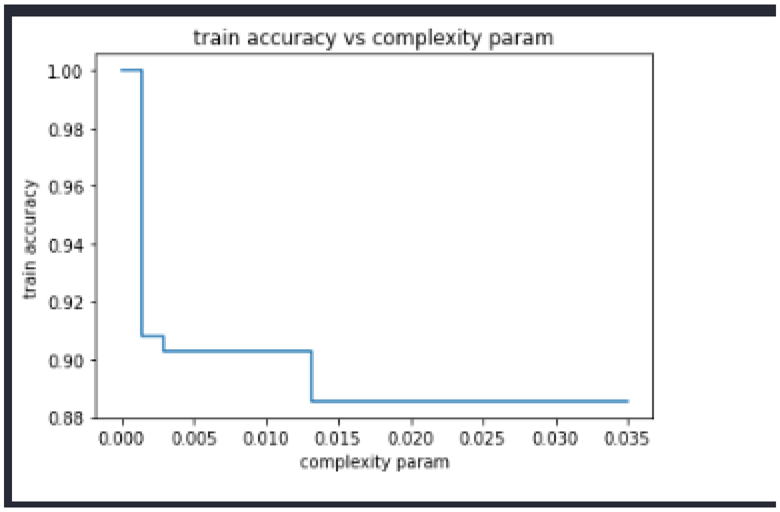
```python
def accuracy(actual, pred):
    assert (len(actual) == len(pred))
    correct = sum([(actual[i] == pred[i]) for i in range(len(actual))])
    return correct / len(actual)
```

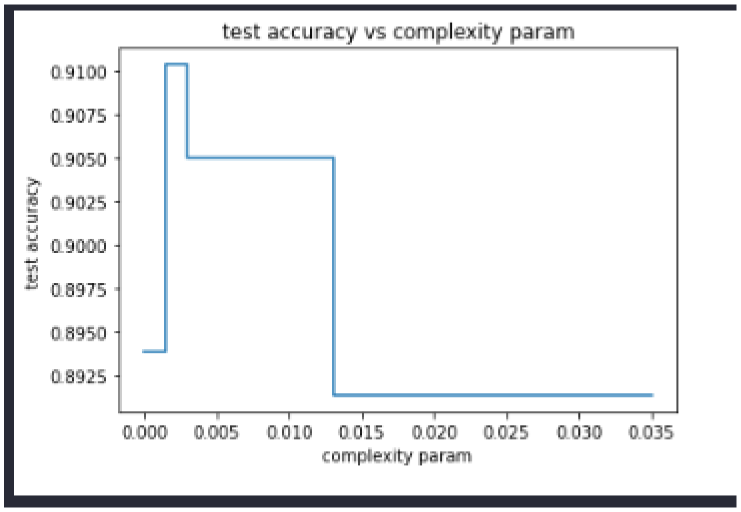
6. **[2.1]** For calculating the impurity of leaf nodes of a decision tree, traversal of the tree is done where leaf nodes are identified (left and right child are same) and a sum of their sample weighted/unweighted impurities are returned.



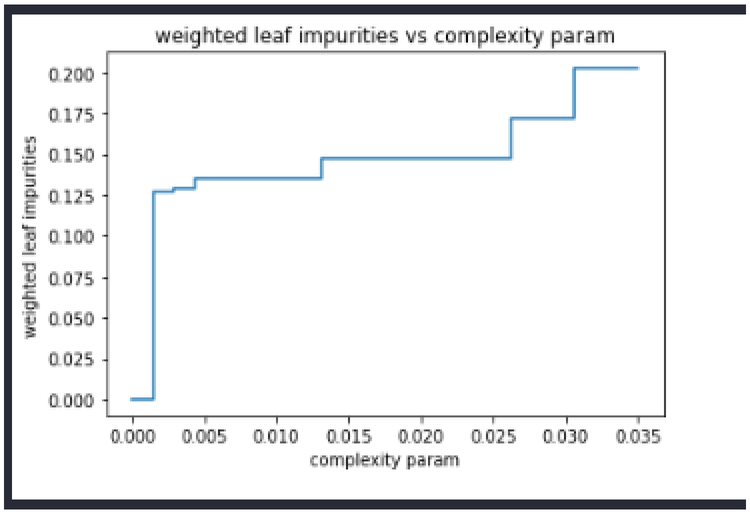
```python
# returns total samples impurity of terminal nodes
def impurity_leaf_nodes(clf: DecisionTreeClassifier, weighted=True):
    n = clf.tree_.node_count
    is_leaf = [(clf.tree_.children_left[i] == clf.tree_.children_right[i]) for i in range(n)]
    total_impurity = 0
    total_impurity_unweighted = 0
    num_samples = 0
    for i in range(n):
        if is_leaf[i]:
            total_impurity += (clf.tree_.n_node_samples[i] * clf.tree_.impurity[i])
            total_impurity_unweighted += clf.tree_.impurity[i]
            num_samples += clf.tree_.n_node_samples[i]
    return (total_impurity / num_samples) if weighted else total_impurity_unweighted
```

7. **[2.1]** Taking complexity parameter (ccp_alpha) as hyperparameter, we perform grid search over 25 values in a linspace of 0 → 0.035. For each such complexity parameter, we calculate impurity of leaf nodes (using our method), train & test acc.

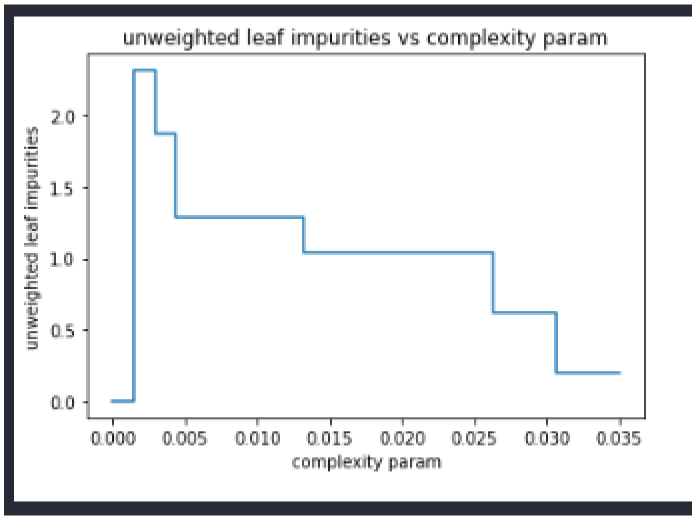




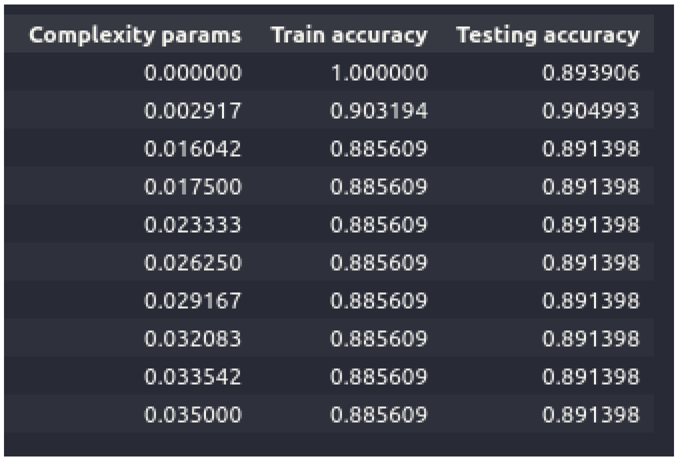
A complexity parameter is used to prune the decision tree until the minimal effective complexity value for all nodes is greater than the given complexity parameter. With the aforementioned, this pruning helps avoid overfitting. Thus initially the tree is overfitted (resulting in low accuracy), which then transcends into a sweet spot with ccp_alpha ≈ 0.003 where the classifier performs the best, which then with further increase due to excessive pruning starts underfitting (around ~0.012), resulting again in a decrease in accuracy. In accordance with the below graph for weighted leaf impurities, we see that as ccp_alpha increases, we notice increase in impurity at leaf nodes, which is due to pruning as in general in a tree for a subtree, the total impurity at leaf nodes is less than impurity of subtree node (misclassification rate). Thus, as we prune these subtree roots, they become leaves, and the effective total impurity increases.



Although in the unweighted leaf impurities graph, the inference is that impurities are highest at ~0.003 but reasoning is hard as the sole sum of impurities is just a sum of ratios which doesn't convey anything.



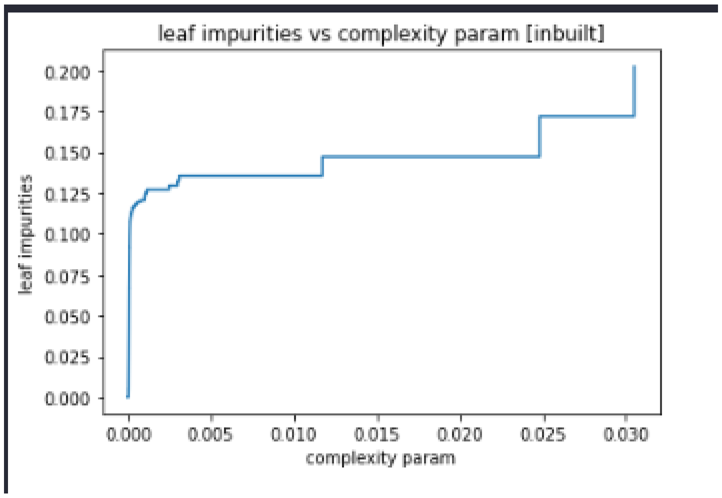
8. **[2.2]** Table representing complexity param, training and testing accuracy (random 10 values of 25)

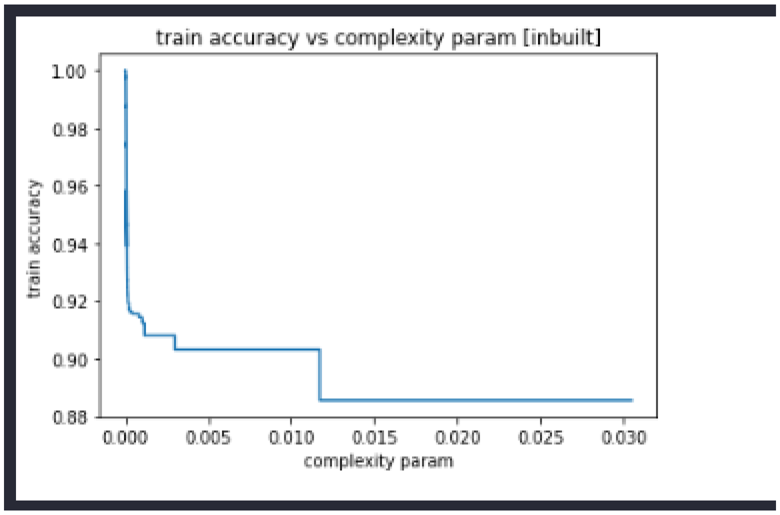


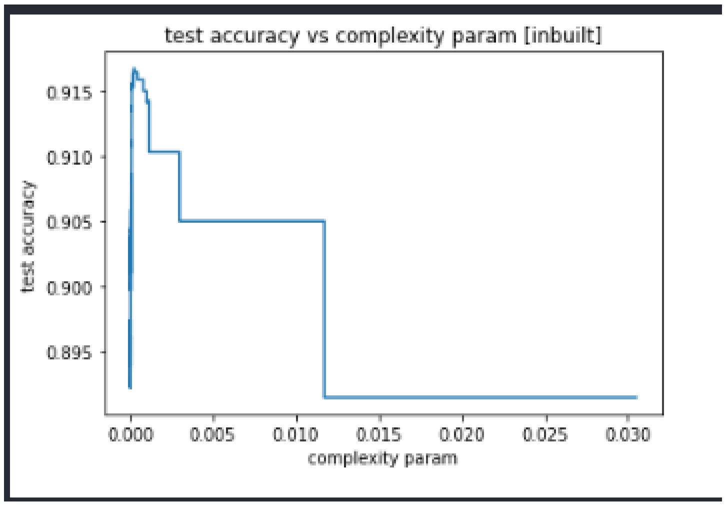
Complexity params	Train accuracy	Testing accuracy
0.000000	1.000000	0.893906
0.002917	0.903194	0.904993
0.016042	0.885609	0.891398
0.017500	0.885609	0.891398
0.023333	0.885609	0.891398
0.026250	0.885609	0.891398
0.029167	0.885609	0.891398
0.032083	0.885609	0.891398
0.033542	0.885609	0.891398
0.035000	0.885609	0.891398

Comments:
First entry clearly showcases overfitting as there is no pruning occurring which is visible through 100% train accuracy and 0.89 test accuracy. The second entry showcases good fitting as training and testing accuracy are significantly high and similar. Rest of the entries showcase underfitting, as at higher ccp_alphas, unnecessary nodes are getting pruned, resulting in improper training, and relatively low testing accuracy. Kindly note that as the class imbalance is of 9:1, accuracy of 0.88 on training and 0.89 on testing is low which shows underfitting.

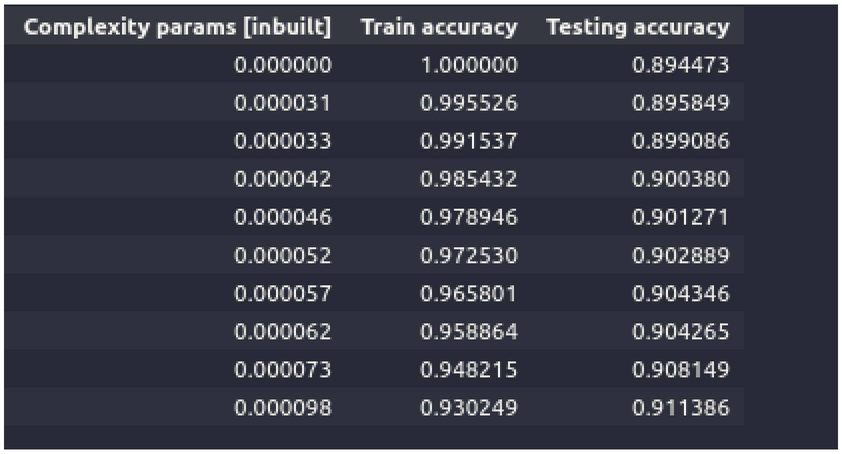
9. **[2.3]** Comparing results from sklearn Decision Tree Classifier's cost_complexity_pruning_path, no/slight deviation between the results w.r.t to the nature of the graphs for weighted impurity, training and testing accuracy is observed.



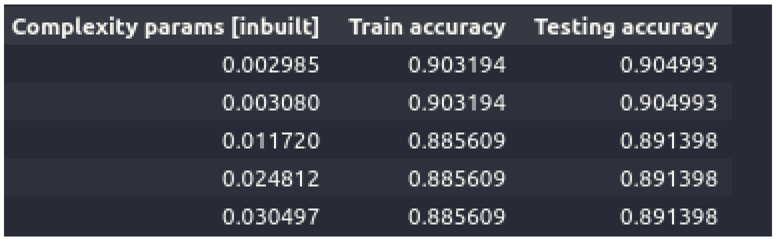




10. **[2.3]** Table representing complexity param [inbuilt], training and testing accuracy (linspaced 10 values) & (tail values):



Complexity params [inbuilt]	Train accuracy	Testing accuracy
0.000000	1.000000	0.894473
0.000031	0.995526	0.895849
0.000033	0.991537	0.899086
0.000042	0.985432	0.900380
0.000046	0.978946	0.901271
0.000052	0.972530	0.902889
0.000057	0.965801	0.904346
0.000062	0.958864	0.904265
0.000073	0.948215	0.908149
0.000098	0.930249	0.911386



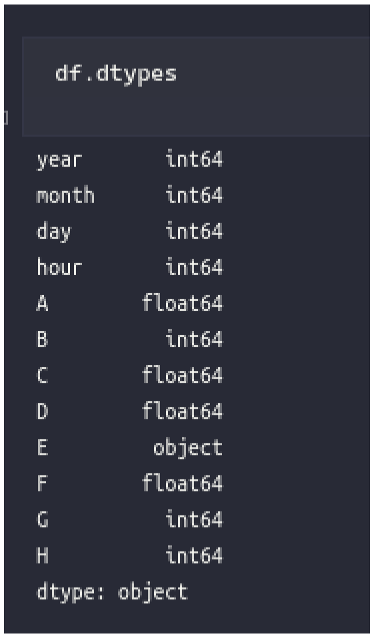
Complexity params [inbuilt]	Train accuracy	Testing accuracy
0.002985	0.903194	0.904993
0.003080	0.903194	0.904993
0.011720	0.885609	0.891398
0.024812	0.885609	0.891398
0.030497	0.885609	0.891398

Comments:
The inbuilt results are quite similar to above manual implemented results. Although, the former table, as the inbuilt ccp_alphas are heavily concentrated in the around and before the optimal alpha region, showcase overfitting. The latter table (the tail values), the first 2 rows showcase good fitting while the last 3 showcase underfitting with alpha values & results similar to the manual alpha values results. The inference and explanation is similar as aforementioned.

Q3.

Dataset: dataset_3.csv
Approach:

1. .dtypes of the data frame to check for categorical and numerical columns



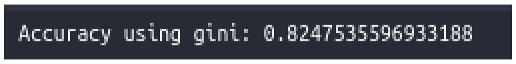
```
df.dtypes
```

```
year       int64
month      int64
day        int64
hour       int64
A        float64
B          int64
C        float64
D        float64
E         object
F        float64
G          int64
H          int64
dtype: object
```

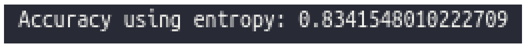
2. **Preprocessing**:
    a. Column A only contained null values, which are handled via two ways
        i. Sampling from a gaussian of existing distribution
        ii. Filling 0

    The latter slightly performed better (~1% accuracy), thus was chosen.

   b. As Column E was categorical with no inferable ordinality, the column is one hot encoded.
   c. The column 'No' was dropped, as it was a serial number for each data entry.
3. The same split function was used as from Q2 to split into training and testing. Also the same accuracy function was used from Q2 to calculate accuracy.
4. **[3.1]** After splitting, two separate decision trees with different criterias i.e. gini & entropy were tested on the test set. Entropy as a criterion was chosen as it performed slightly better than gini (~1% accuracy).

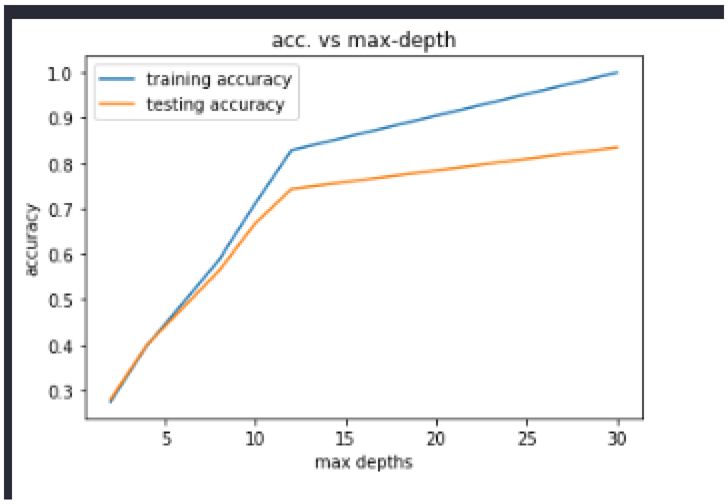


```
Accuracy using gini: 0.8247535596933188
```



```
Accuracy using entropy: 0.8341548010222709
```

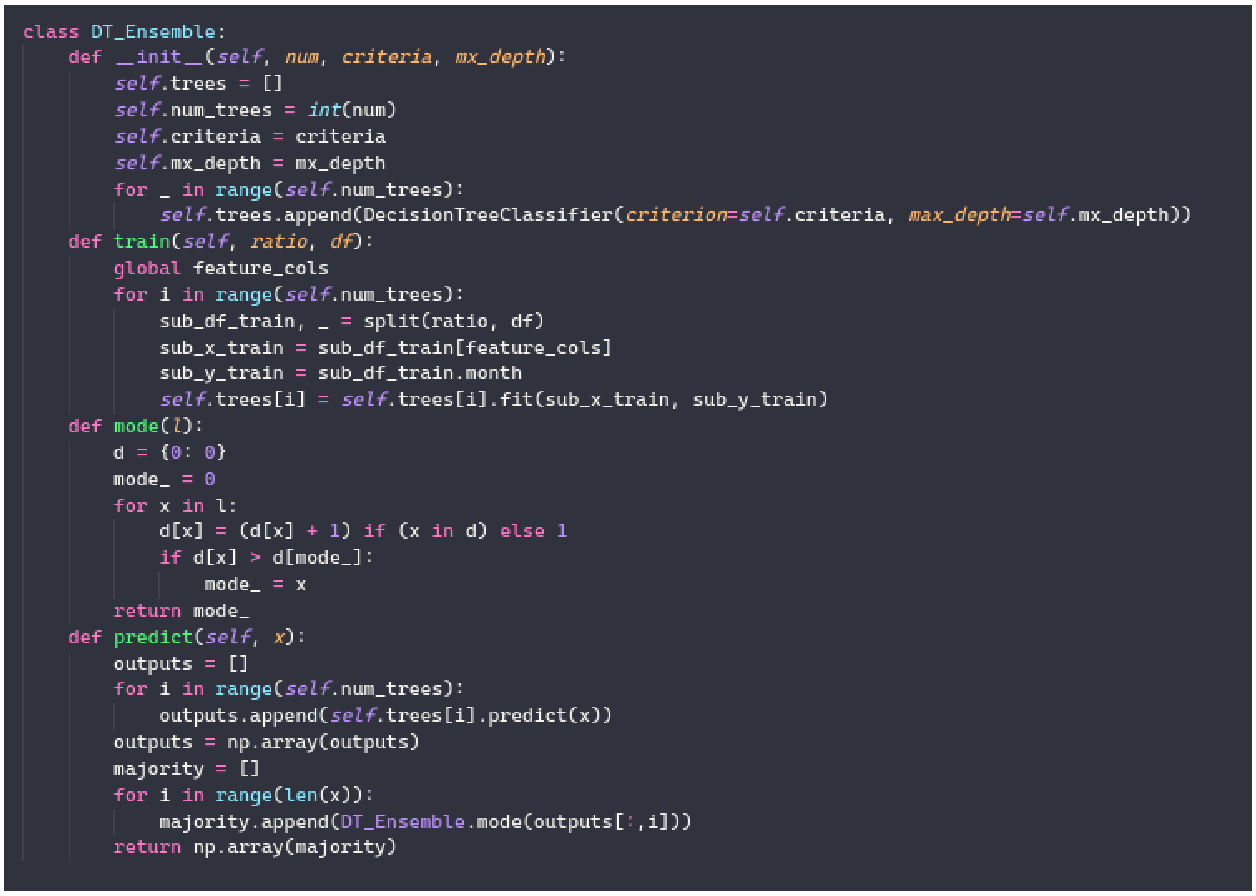
5. **[3.2]** With maximum depths of [2, 4, 8, 10, 12, 30], the following graph is observed



   Beyond max depth of around 13, there is a sharp change in gradient, and training and testing accuracy start to diverge (with much steep gradient for testing). Although, at depth 30, the testing accuracy still seems to gradually increase, thus it is chosen as the depth with the maximum accuracy.
6. **[3.3]** Ensemble of 150 decision trees is created with a max depth of 4 where each stump is provided with 40% of training data. Finally a majority vote of the outputs is taken for prediction.
   On training, a low accuracy of ~0.4 is observed which performs much worse in comparison to previous decision trees (with higher depth). Compared to decision trees with lower depth, despite the large ensemble size, we only get a very slight increase in accuracy.

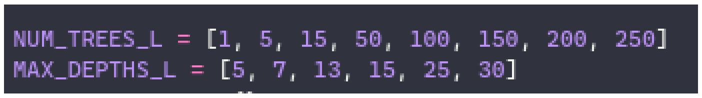
- Decision tree with max depth 4 (Accuracy): ~0.35%
- Ensemble with max depth 4, size 150 and 40% training data per stump (Accuracy): ~0.4%

A major reason for this observation is due to the limitation of max depth, where the tree is not able to fully explore the complex input space. Thus, despite the high ensemble size, we only get a very slight increase in accuracy.



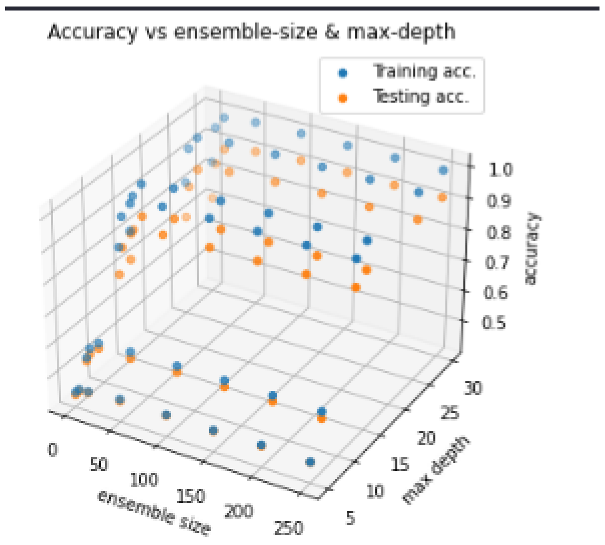
```python
class DT_Ensemble:
    def __init__(self, num, criteria, mx_depth):
        self.trees = []
        self.num_trees = int(num)
        self.criteria = criteria
        self.mx_depth = mx_depth
        for _ in range(self.num_trees):
            self.trees.append(DecisionTreeClassifier(criterion=self.criteria, max_depth=self.mx_depth))
    def train(self, ratio, df):
        global feature_cols
        for i in range(self.num_trees):
            sub_df_train, _ = split(ratio, df)
            sub_x_train = sub_df_train[feature_cols]
            sub_y_train = sub_df_train.month
            self.trees[i] = self.trees[i].fit(sub_x_train, sub_y_train)
    def mode(l):
        d = {0: 0}
        mode_ = 0
        for x in l:
            d[x] = (d[x] + 1) if (x in d) else 1
            if d[x] > d[mode_]:
                mode_ = x
        return mode_
    def predict(self, x):
        outputs = []
        for i in range(self.num_trees):
            outputs.append(self.trees[i].predict(x))
        outputs = np.array(outputs)
        majority = []
        for i in range(len(x)):
            majority.append(DT_Ensemble.mode(outputs[:,i]))
        return np.array(majority)
```

7. **[3.4]** Ensemble w/ varying hyperparameters:
   a. The following hyperparameters were tested:



```python
NUM_TREES_L = [1, 5, 15, 50, 100, 150, 200, 250]
MAX_DEPTHS_L = [5, 7, 13, 15, 25, 30]
```

   b. The following plot was observed:



For a certain max depth, the accuracies both on training and testing increase upto a certain threshold with increase in ensemble size, but with further increase, only minimal changes in the accuracy is observed. As max depth increases, performance over both test and train set increases, although the test set's gradient seems to gradually decrease (still increasing due to regularization), which occurs due to introduction of overfitting on the training data. A significantly higher max depth with lower ensemble sizes (above a certain threshold i.e. ~15) performs much better than lower max depth with higher ensemble sizes. With lower ensemble sizes, a dip in testing accuracy & training accuracy is observed due to underfitting (for ex. ensemble size of 1 or 5). This trend is not observed in higher ensemble sizes. Rather a gradual increase in testing accuracy with max depths is observed which showcases regularization properties due to majority vote and different training data for each stump.

8. **[Ranking]** Ranking the models based on test accuracies [best to worst]: (Considering training ratio of each stump to be 0.4)
   a. Ensemble with size $S > S_T$ and depth D
   b. Single decision tree with depth D
   c. Ensemble with size $S > S_T$ and depth $D_1 << D$
   d. Single decision tree with depth $D_1 << D$
   e. Ensemble with size S_T or lower with depth D
   f. Ensemble with size S_T or lower with depth $D_1 << D$

There is a statistically significant difference as we go from b $\rightarrow$ c & d $\rightarrow$ e. The former is due to unnecessary pruning which avoids the decision tree to explore the complex input space completely. The latter is due to the threshold size for ensemble. S_T is a threshold size for the ensemble, which in the current case is taken as ~5. This threshold signifies lesser provision of training data (considering 0.4 ratio) for each stump and given the high class imbalance, it is highly likely that the minority class could be considered as outliers for stumps resulting in bad performance.