orszufbdm

October 22, 2023

For the second question in the Assignment, I have chosen “bank.csv”, A dataset that helps predicts if a customer decides to open up a term deposit.

What is a term deposit? A Term deposit is a deposit that a bank or a financial institurion offers with a fixed rate (often better than just opening deposit account) in which your money will be returned back at a specific maturity time.

Source: Investopedia: https://www.investopedia.com/terms/t/termdeposit.asp

Here are the features present in the dataset: 1 - age: (numeric)

2 - job: type of job (categorical: ‘admin.’,‘blue-collar’,‘entrepreneur’,‘housemaid’,‘management’,‘retired’,‘self-employed’,‘services’,‘student’,‘technician’,‘unemployed’,‘unknown’)

3 - marital: marital status (categorical: ‘divorced’,‘married’,‘single’,‘unknown’; note: ‘divorced’ means divorced or widowed)

4 - education: (categorical: primary, secondary, tertiary and unknown)

5 - default: has credit in default? (categorical: ‘no’,‘yes’,‘unknown’)

6 - housing: has housing loan? (categorical: ‘no’,‘yes’,‘unknown’)

7 - loan: has personal loan? (categorical: ‘no’,‘yes’,‘unknown’)

8 - balance: Balance of the individual.

9 - contact: contact communication type (categorical: ‘cellular’,‘telephone’)

10 - month: last contact month of year (categorical: ‘jan’, ‘feb’, ‘mar’, …, ‘nov’, ‘dec’)

11 - day: last contact day of the week (categorical: ‘mon’,‘tue’,‘wed’,‘thu’,‘fri’)

12 - duration: last contact duration, in seconds (numeric).

13 - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)

14 - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)

15 - previous: number of contacts performed before this campaign and for this client (numeric)

16 - poutcome: outcome of the previous marketing campaign (categorical: ‘failure’,‘nonexistent’,‘success’)

17:deposit : Out Target variable, which is binary.(yes/no)

```
[ ]: #import the necessary packages
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

0.1 Question 1

```
[ ]: #read our as a pandas data frame
df = pd.read_csv("./bank.csv")
print("-------------------------------------------------------------")
#print the dimension
print("\n The shape of the dataframe is " , df.shape)
print("-------------------------------------------------------------")
#print all columns
print("\n The columns that are present in the dataframe" , df.columns)
#view the first 5 rows of the datafram
print("-------------------------------------------------------------")
print("\n Basic statistical info of the dataset is:\n")
print(df.info())
print("-------------------------------------------------------------")
print("\n The first five rows of the dataframe :" )
df.head(5)
```

```
-------------------------------------------------------------

 The shape of the dataframe is  (11162, 17)
-------------------------------------------------------------

 The columns that are present in the dataframe Index(['age', 'job', 'marital',
'education', 'default', 'balance', 'housing',
       'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays',
       'previous', 'poutcome', 'deposit'],
      dtype='object')
-------------------------------------------------------------

 Basic statistical info of the dataset is:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11162 entries, 0 to 11161
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        11162 non-null  int64
 1   job        11162 non-null  object
 2   marital    11162 non-null  object
 3   education  11162 non-null  object
```

```
 4   default    11162 non-null  object
 5   balance    11162 non-null  int64
 6   housing    11162 non-null  object
 7   loan       11162 non-null  object
 8   contact    11162 non-null  object
 9   day        11162 non-null  int64
 10  month      11162 non-null  object
 11  duration   11162 non-null  int64
 12  campaign   11162 non-null  int64
 13  pdays      11162 non-null  int64
 14  previous   11162 non-null  int64
 15  poutcome   11162 non-null  object
 16  deposit    11162 non-null  object
dtypes: int64(7), object(10)
memory usage: 1.4+ MB
None
------------------------------------------------------------

 The first five rows of the dataframe :
```

[ ]:

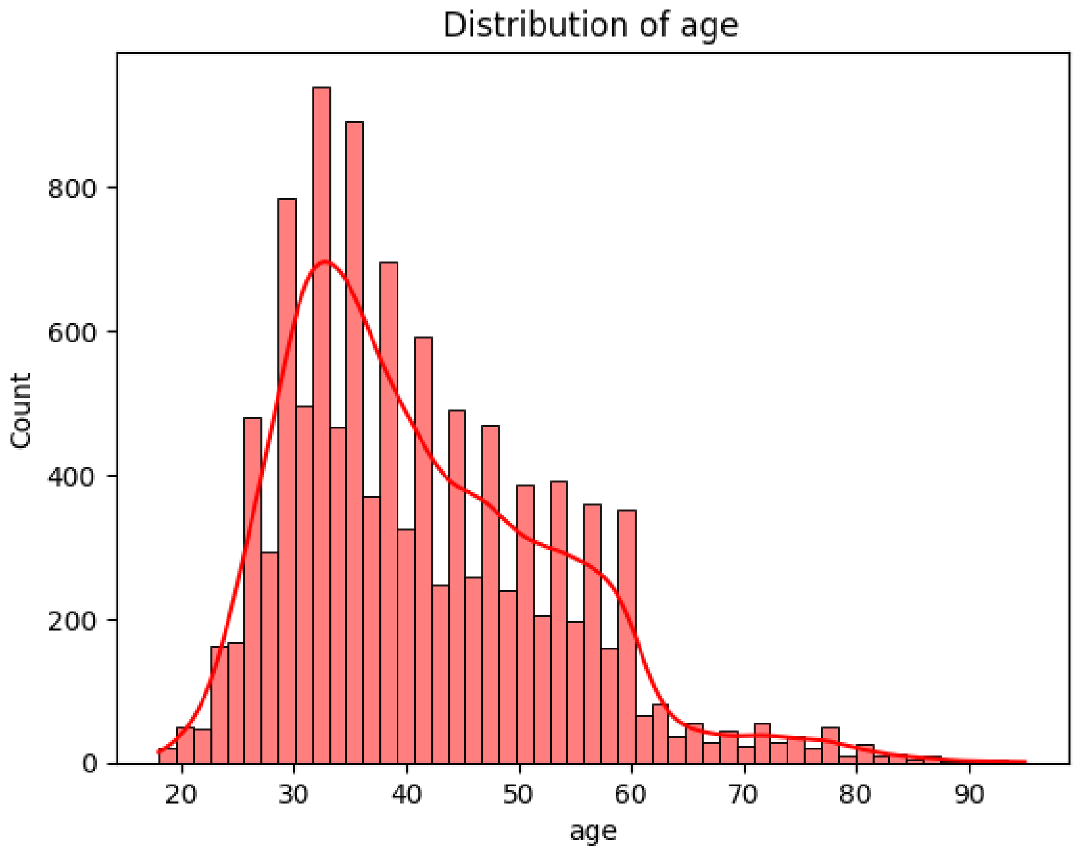
```
   age         job  marital  education default  balance housing loan  contact  \
0   59      admin.  married  secondary      no     2343     yes   no  unknown
1   56      admin.  married  secondary      no       45      no   no  unknown
2   41  technician  married  secondary      no     1270     yes   no  unknown
3   55    services  married  secondary      no     2476     yes   no  unknown
4   54      admin.  married   tertiary      no      184      no   no  unknown

   day month  duration  campaign  pdays  previous poutcome deposit
0    5   may      1042         1     -1         0  unknown     yes
1    5   may      1467         1     -1         0  unknown     yes
2    5   may      1389         1     -1         0  unknown     yes
3    5   may       579         1     -1         0  unknown     yes
4    5   may       673         2     -1         0  unknown     yes
```

[ ]:

```python
#helper function for continuous columns
def draw_plot_univariate_cont(column):
  print(f"Statistical summary of {column}:\n\n",df[column].describe())
  print("----------------------------------")
  print("Missing values: %f\n" % df[column].isnull().sum())
  print("Mean: %f\n" % df[column].mean())
  print("Median: %f \n" % df[column].median())
  print("Skewness: %f\n" % df[column].skew())
  print("Kurtosis: %f\n" % df[column].kurt())
  print("----------------------------------")
  sns.histplot(df[str(column)],color = "red",kde="True").
 ↪set_title(f"Distribution of {column}")
```

[ ]:
```python
# age
draw_plot_univariate_cont("age")
```

```
Statistical summary of age:

 count    11162.000000
mean        41.231948
std         11.913369
min         18.000000
25%         32.000000
50%         39.000000
75%         49.000000
max         95.000000
Name: age, dtype: float64
---------------------------------
Missing values: 0.000000

Mean: 41.231948

Median: 39.000000

Skewness: 0.862780

Kurtosis: 0.621540

---------------------------------
```

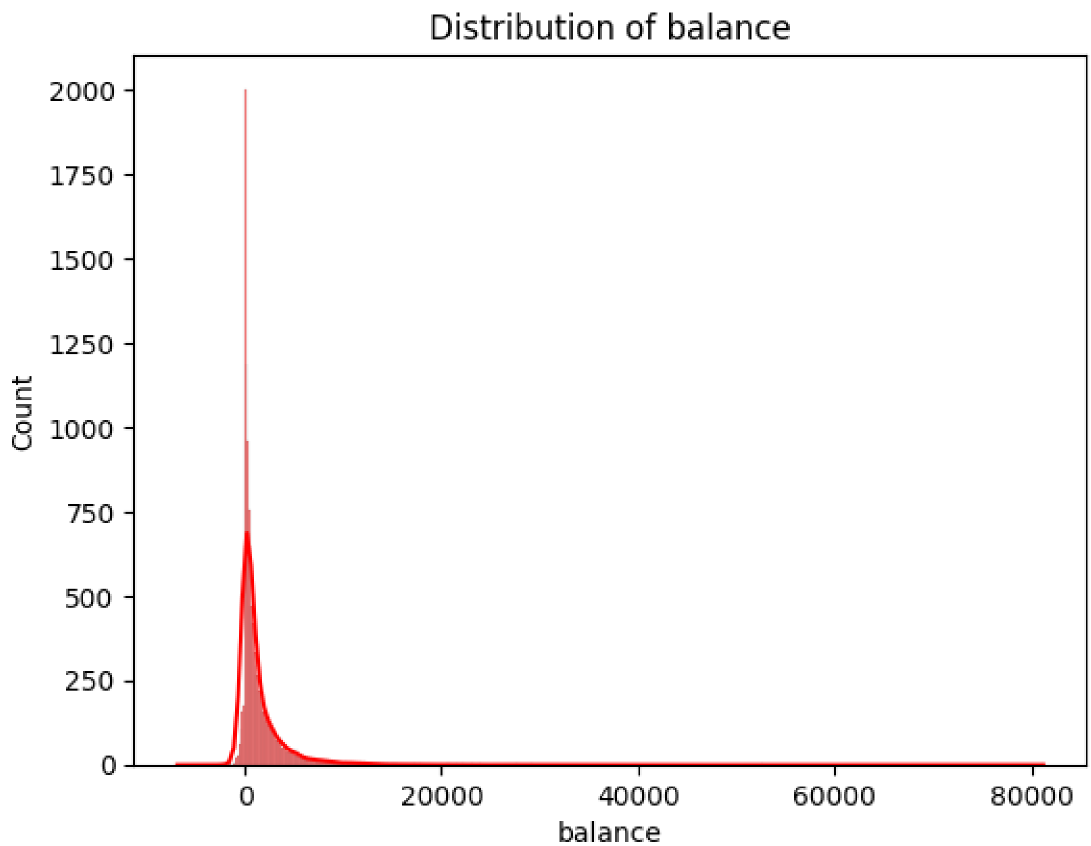


- It looks like the mean age is around 41, and the skewness is 0.82, slightly positively skewed, which means the median will be lesser than the mean.
- Using IQR to remove outliers might help making the distribution more normal, But one can perform log or square root Transformation which has proven to make the data more normal. *There are are no missing values.* Additionally, one can standardize or normalize the data at the cost of interpretability(target column).

[ ]:
```
draw_plot_univariate_cont("balance")
```

```
Statistical summary of balance:

 count    11162.000000
mean      1528.538524
std       3225.413326
min      -6847.000000
25%        122.000000
50%        550.000000
75%       1708.000000
max      81204.000000
Name: balance, dtype: float64
```

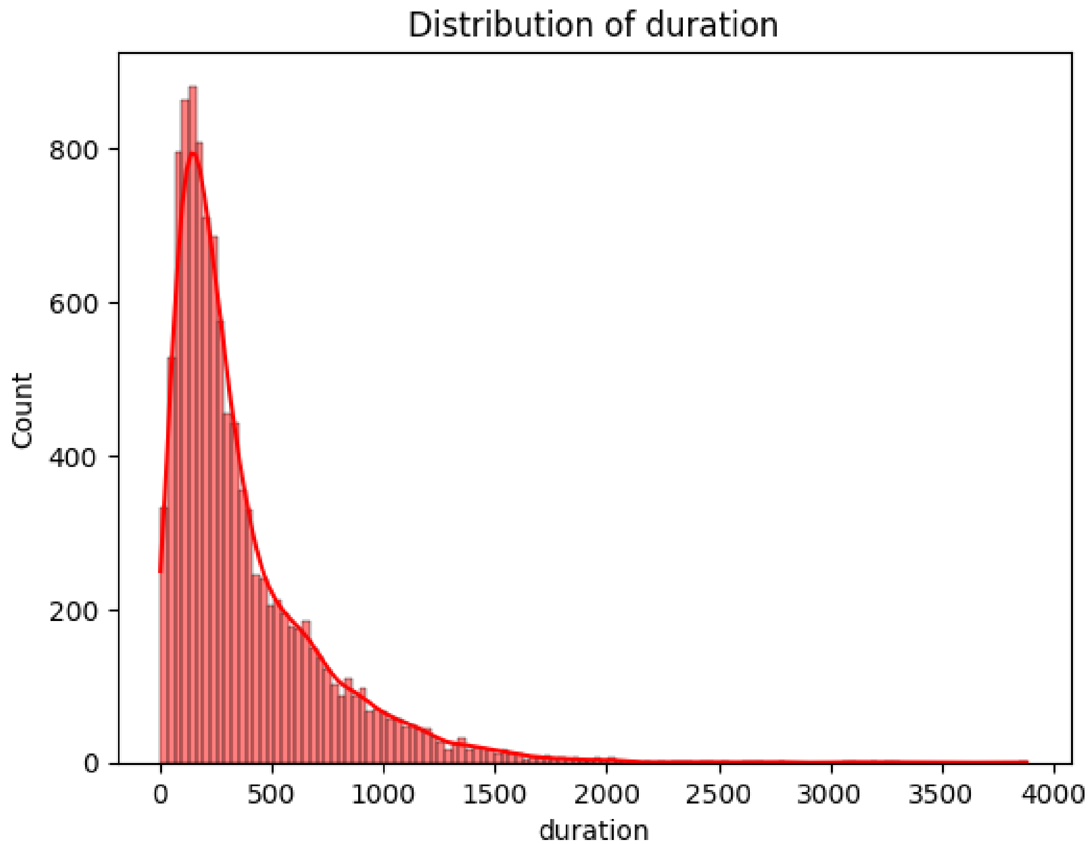
```
---------------------------------
Missing values: 0.000000

Mean: 1528.538524

Median: 550.000000

Skewness: 8.224619

Kurtosis: 126.861303

---------------------------------
```



- It looks like the mean balance is around 1528units, and the skewness is 8.22, which means that the distribition is very highly positively skewed.
- But one can perform log or square root Transformation to make the distributionmore normal.
- There are are no missing values.
- Additionally, one can standardize or normalize the data at the cost of interpretability(target column)

[ ]:
```python
draw_plot_univariate_cont("duration")
```

```
Statistical summary of duration:

 count    11162.000000
mean       371.993818
std        347.128386
min          2.000000
25%        138.000000
50%        255.000000
75%        496.000000
max       3881.000000
Name: duration, dtype: float64
---------------------------------
Missing values: 0.000000

Mean: 371.993818

Median: 255.000000

Skewness: 2.143695

Kurtosis: 7.301282

---------------------------------
```



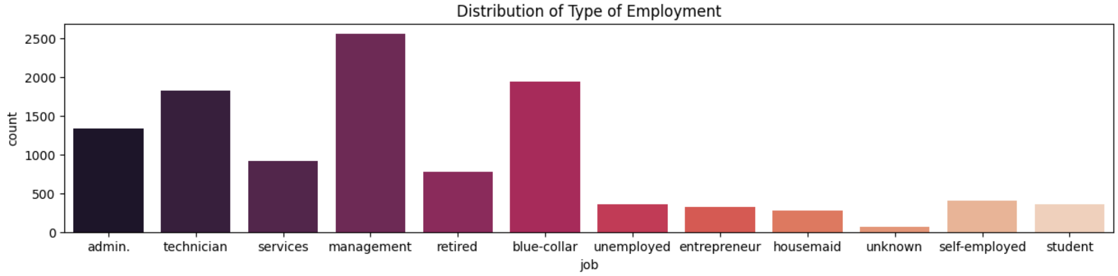
- It looks like the mean duration of the call is around 371 seconds, and the skewness is 2.14, which means that the distribition is v positively skewed.
- one can perform log , square root,or inverse Transformation to make the distributionmore normal.
- There are are no missing values.
- Additionally, one can standardize or normalize this continuos data at the cost of interpretability(target column)

[ ]:
```
#plot for categorical columns
def draw_plot_univariate_cat(column,plot_description,fig_x=5,fig_y=3):
  print(f"Statistical summary of {column}:\n\n", df[column].
 ↪describe(include='object'))
  print("-----------------------------------")
  print("Missing values: %f\n" % df[column].isnull().sum())
  print("-----------------------------------")
  print("Count")
  print("-----------------------------------")
  print(df[column].value_counts())
  print("-----------------------------------")
  plt.figure(figsize=(fig_x,fig_y))
```

```
    ax = sns.countplot(x=column, data=df,palette="rocket").
↪set_title(plot_description)
    plt.show()
```

```
[ ]: draw_plot_univariate_cat("job","Distribution of Type of Employment",15,3)
```

```
Statistical summary of job:

 count          11162
unique             12
top        management
freq             2566
Name: job, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
management       2566
blue-collar      1944
technician       1823
admin.           1334
services          923
retired           778
self-employed     405
student           360
unemployed        357
entrepreneur      328
housemaid         274
unknown            70
Name: job, dtype: int64
---------------------------------
```

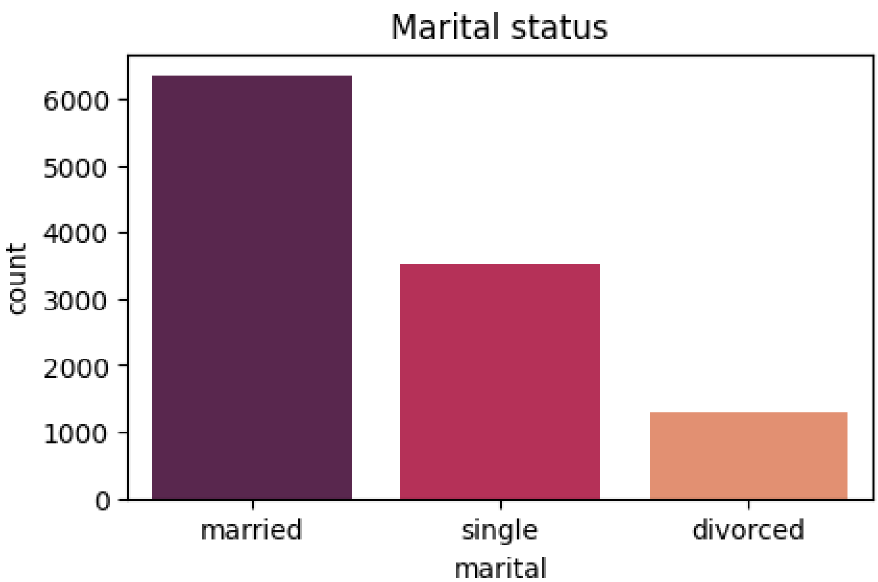


- We have varied distribution of jobs under job column, with 12 differernt kinds of jobs, where management and blue collar jobs top the list.

- These have no order, hence this variable needs to be one hot encoded.

```
[ ]: draw_plot_univariate_cat("marital","Marital status")
```

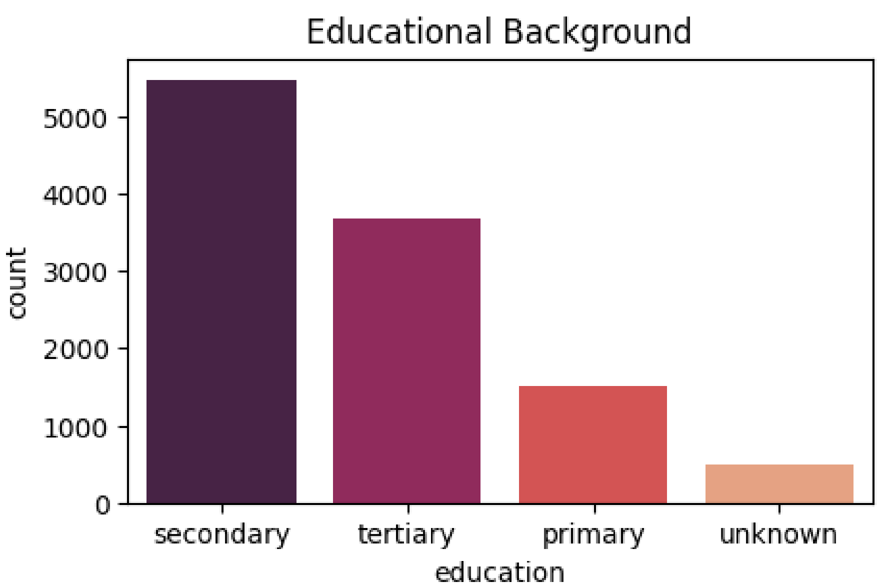
```
Statistical summary of marital:

 count       11162
unique          3
top       married
freq         6351
Name: marital, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
married     6351
single      3518
divorced    1293
Name: marital, dtype: int64
---------------------------------
```



- We have 6351 married people, 3518 single and 1293 divorced people. Further analysis on how this matters in opening a term deposit might help us add weightage for this variable while prediction.
- These have no order, hence this variable needs to be one hot encoded.

[ ]:
```
draw_plot_univariate_cat("education","Educational Background",5,3)
```

```
Statistical summary of education:

 count          11162
unique              4
top         secondary
freq             5476
Name: education, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
secondary    5476
tertiary     3689
primary      1500
unknown       497
Name: education, dtype: int64
---------------------------------
```

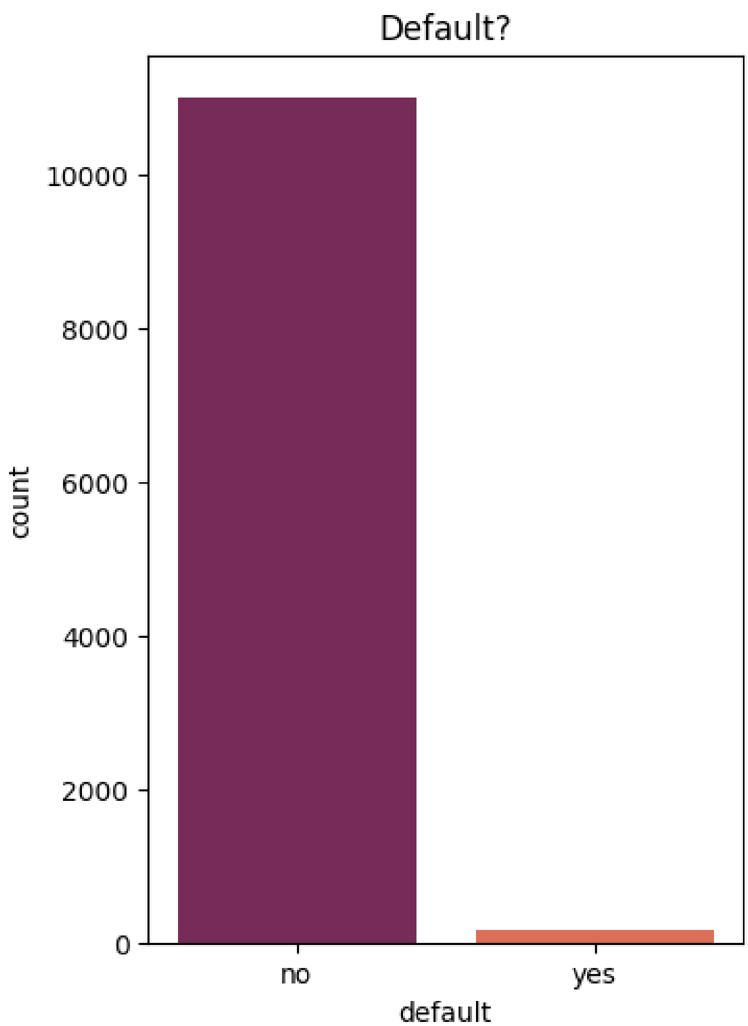


- Close to half the people have secondary education. Because this is a ordinal variable, label encoding should be done.

[ ]:
```
draw_plot_univariate_cat("default","Default?",4,6)
```

```
Statistical summary of default:
```

```
 count     11162
unique         2
top           no
freq       10994
Name: default, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
no     10994
yes      168
Name: default, dtype: int64
---------------------------------
```



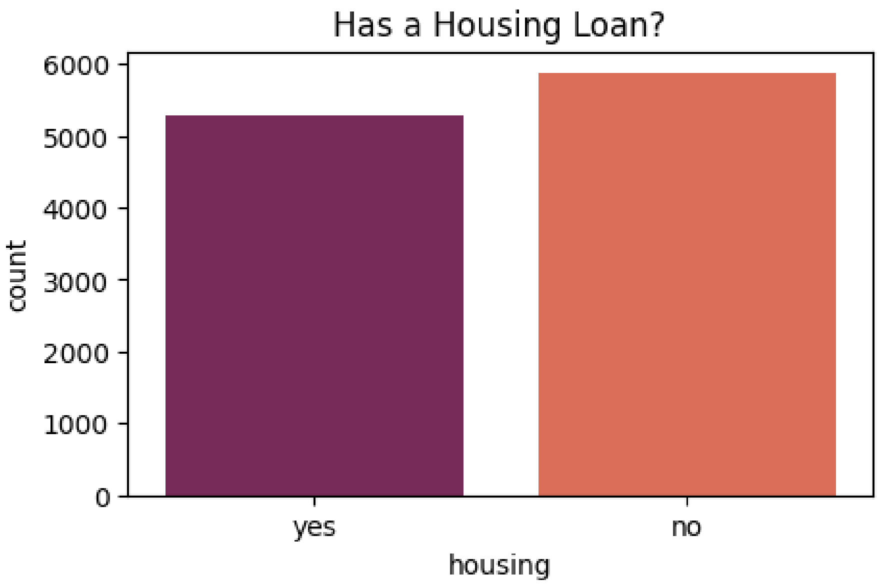
[ ]:
```
df["default"].value_counts()/(len(df))
```

[ ]:
```
no     0.984949
yes    0.015051
Name: default, dtype: float64
```

- This is an Highly imbalanced variable with close to 98% of the people with no defaults.
- while splitting the dataset, we gotta make sure that this variable is stratified considering how much correlated it is with the Target variable.
- Binary encoding is the way to go here

[ ]:
```
draw_plot_univariate_cat("housing","Has a Housing Loan?")
```

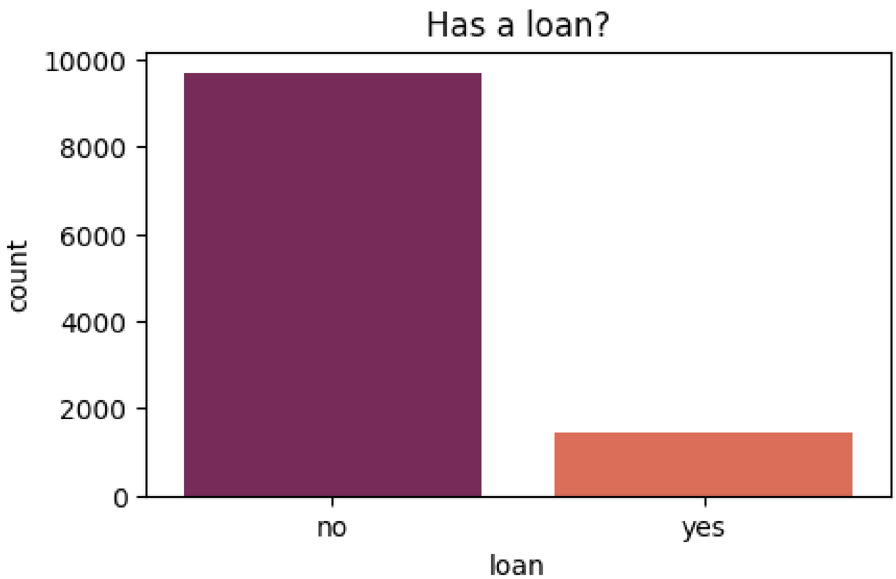
```
Statistical summary of housing:

 count     11162
unique        2
top          no
freq       5881
Name: housing, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
no     5881
yes    5281
Name: housing, dtype: int64
---------------------------------
```



- This variable is more or less balanced.
- Should be one hot encoded as there is no order in them

[ ]:
```
draw_plot_univariate_cat("loan","Has a loan?")
```

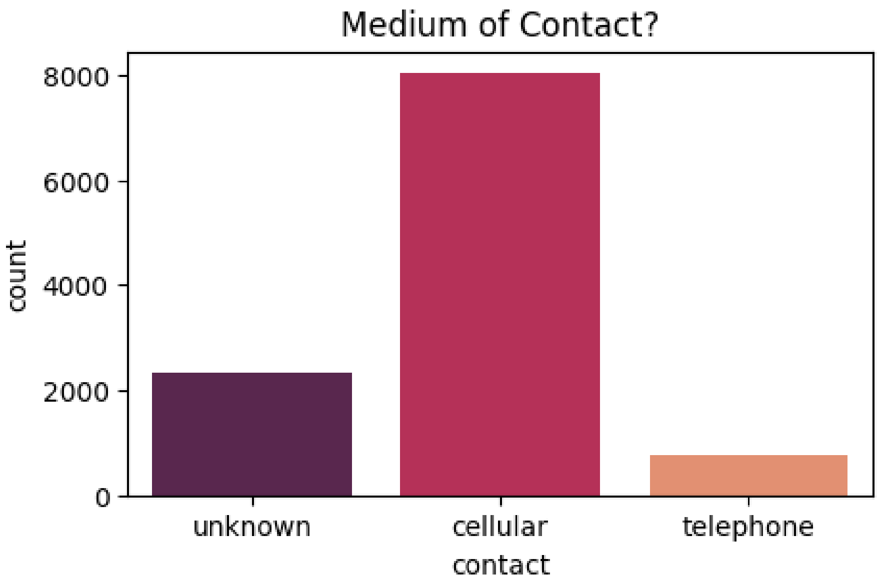
```
Statistical summary of loan:

 count     11162
unique        2
top          no
freq       9702
Name: loan, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
no     9702
yes    1460
Name: loan, dtype: int64
---------------------------------
```



- Most of the people do not have a loan, whereeas close to half of them have house loan.
- Similar to default variable, loan variable is also imbalanced.

[ ]:
```
draw_plot_univariate_cat("contact","Medium of Contact?")
```

```
Statistical summary of contact:

 count        11162
unique            3
top        cellular
freq           8042
Name: contact, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
cellular     8042
unknown      2346
telephone     774
Name: contact, dtype: int64
---------------------------------
```

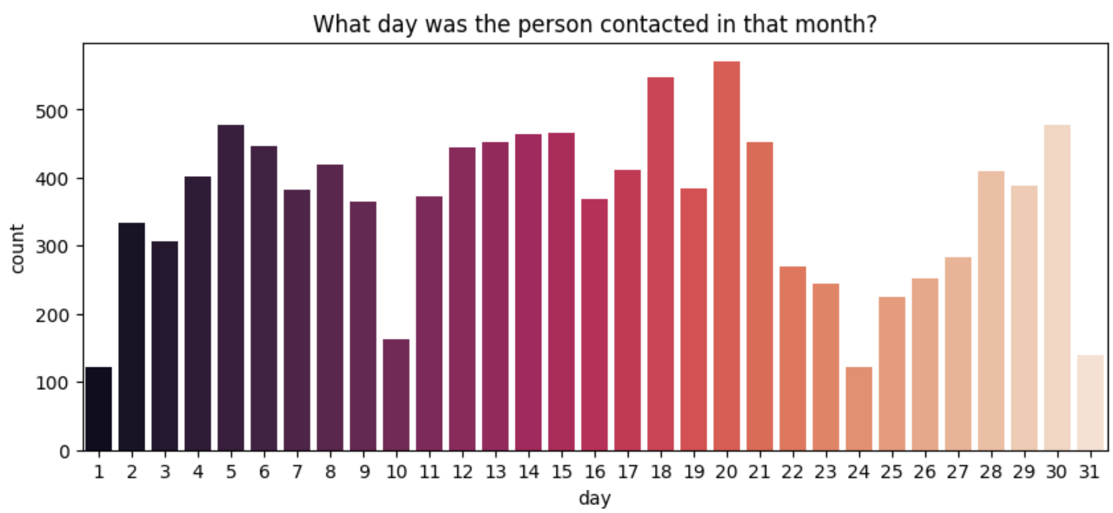


- The customers were majory contacted through the cellphones
- is there an order in this? I personally do not think so. Hence we can go ahead with one hot encoding.

[ ]:
```
draw_plot_univariate_cat("day","What day was the person contacted in that month?
↪",10,4)
```

```
Statistical summary of day:

 count    11162.000000
mean        15.658036
std          8.420740
min          1.000000
25%          8.000000
50%         15.000000
75%         22.000000
max         31.000000
Name: day, dtype: float64
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
20     570
18     548
30     478
5      477
```

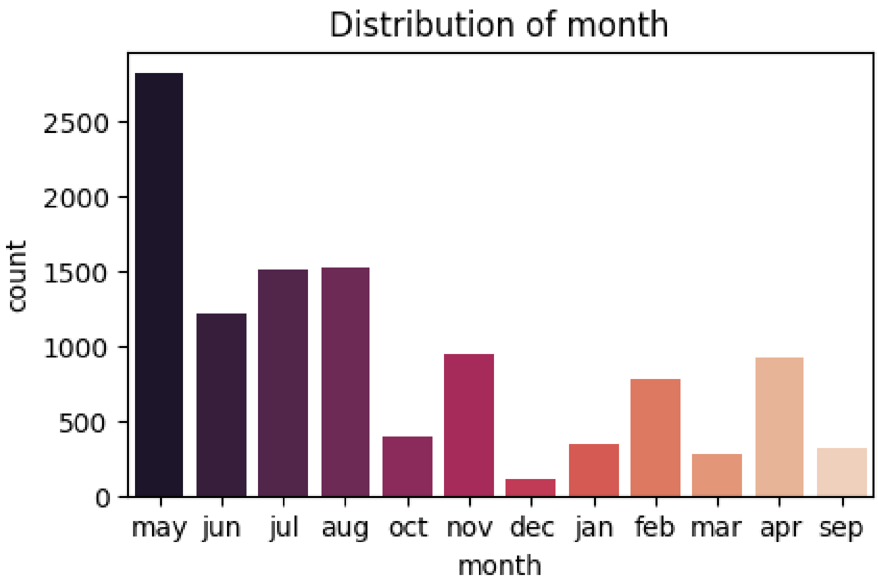
```
15     466
14     463
13     453
21     452
6      447
12     445
8      419
17     411
28     410
4      402
29     388
19     384
7      382
11     373
16     369
9      364
2      334
3      306
27     284
22     269
26     252
23     245
25     224
10     163
31     140
24     122
1      122
Name: day, dtype: int64
--------------------------------
```



- Customers were contacted throught the month, with numbers slightly decreasing towards the end of the month.
- no encoding is required, as it is already in the form of label encoded variable

[ ]:
```python
draw_plot_univariate_cat("month","Distribution of month")
```

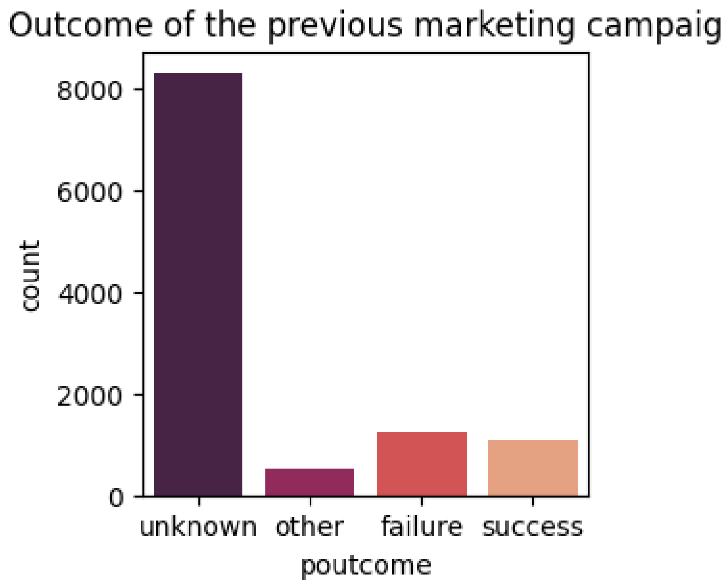
```
Statistical summary of month:

 count     11162
unique       12
top         may
freq       2824
Name: month, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
may    2824
aug    1519
jul    1514
jun    1222
nov     943
apr     923
feb     776
oct     392
jan     344
sep     319
mar     276
dec     110
Name: month, dtype: int64
---------------------------------
```



- For some reason, customers the number of calls made in May is significantly higher than any other month in the entire year.
- One hot encoding or Label encoding? I am going ahead with one hot encoding as i dont see an order in the months.

[ ]:
```
draw_plot_univariate_cat("poutcome","Outcome of the previous marketing␣
↪campaig",3,3)
```

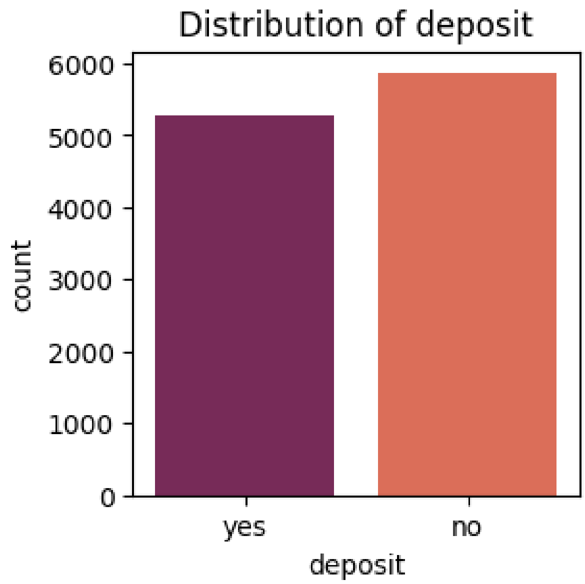
```
Statistical summary of poutcome:

 count       11162
unique          4
top       unknown
freq         8326
Name: poutcome, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
unknown    8326
failure    1228
success    1071
other       537
Name: poutcome, dtype: int64
---------------------------------
```



- Most of the outcomes of the previous marketing campaign on the customers were unknown.
- But, equal possibility of failure and success is also present

[ ]:
```
draw_plot_univariate_cat("deposit","Distribution of deposit",3,3)
```

```
Statistical summary of deposit:

 count     11162
unique        2
top          no
freq       5873
Name: deposit, dtype: object
---------------------------------
Missing values: 0.000000

---------------------------------
Count
---------------------------------
no     5873
yes    5289
Name: deposit, dtype: int64
---------------------------------
```

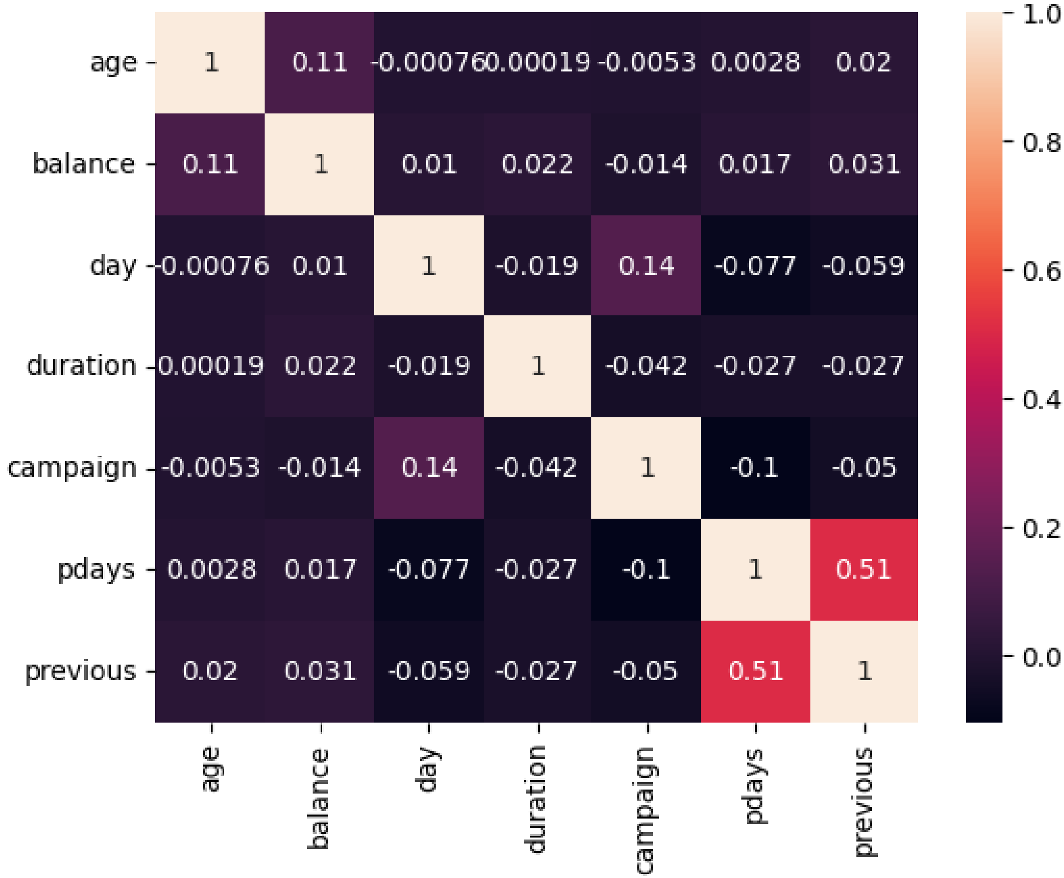


- Finally, we do have almost equal distribution of yes/no target variable "deposit" in our dataset. Hence we dont have to upsample or downsample. or even augment our data to balance the dataset.
- Like always, for target variables, i am going ahead with Label Encoding

Question 2

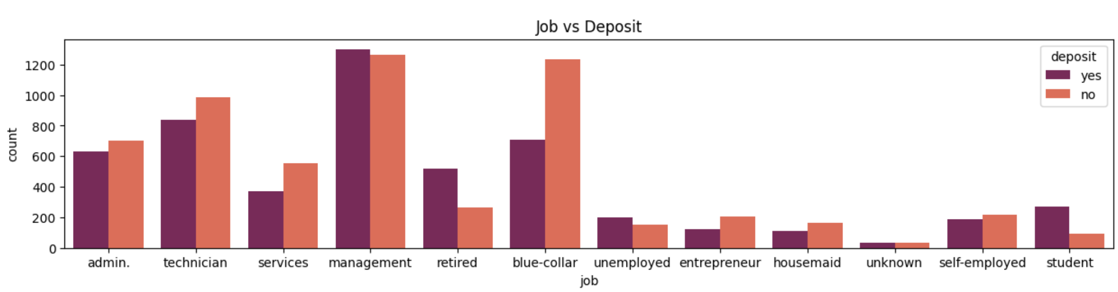
[ ]:
```
sns.heatmap(df.corr(numeric_only=True), annot=True)
```

[ ]: <Axes: >



- For only numerical variables, the correlation matrix does npt display of any direct correlation within each other apart from " pdays and previous".
- Because most of our featues are categorical, we would need further bivariate and multivariate analysis and categorical to numerical correlations calculated to understand the entire dataframe correlation.

[ ]:
```
plt.figure(figsize=(15,3))
sns.countplot(data =df,x= df["job"],hue="deposit",palette="rocket").
  ↪set_title("Job vs Deposit")
plt.show()
```

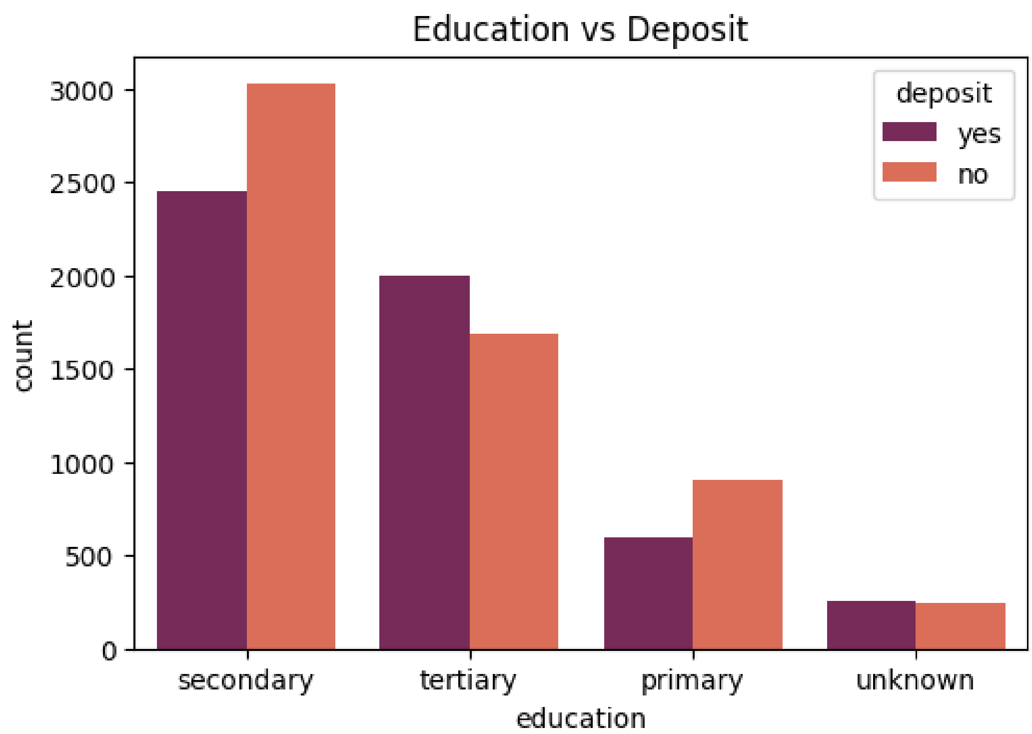


[ ]:
```
df.groupby(["job"]).deposit.value_counts().unstack()
```

[ ]:
```
deposit          no   yes
job
admin.          703   631
blue-collar    1236   708
entrepreneur    205   123
housemaid       165   109
management     1265  1301
retired         262   516
self-employed   218   187
services        554   369
student          91   269
technician      983   840
unemployed      155   202
unknown          36    34
```

- Looks like customers in management have the highest number of conversions, and also at the same time highest number of declines, which makes sense as most of the calls would be made to them.
- unemployed and students have higher number of “YES” than “No”.

[ ]:
```
plt.figure(figsize=(6,4))
sns.countplot(data =df,x= df["education"],hue="deposit",palette="rocket").
↪set_title("Education vs Deposit")
plt.show()
```

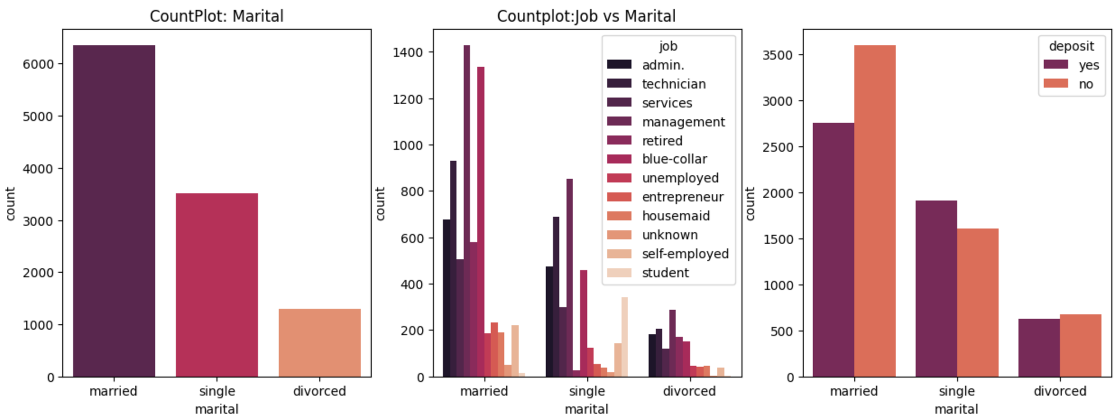


Customers with Tertiary education have higher number of “YES” than “No”.

[ ]:
```
df.groupby(["education"]).deposit.value_counts().unstack()
```

[ ]:
```
deposit      no   yes
education
primary     909   591
secondary  3026  2450
tertiary   1693  1996
unknown     245   252
```

[ ]:
```
fig,axs = plt.subplots(1,3,figsize=(15,5))
sns.countplot(data =df,x=df["marital"],palette="rocket",ax=axs[0]).
 ↪set_title("CountPlot: Marital")
sns.countplot(data =df,x=df["marital"],hue="job",palette="rocket",ax=axs[1]).
 ↪set_title("Countplot:Job vs Marital")
sns.countplot(data =df,x=df["marital"],hue="deposit",palette=␣
 ↪"rocket",ax=axs[2])
plt.subplots_adjust(hspace =0.7)
```

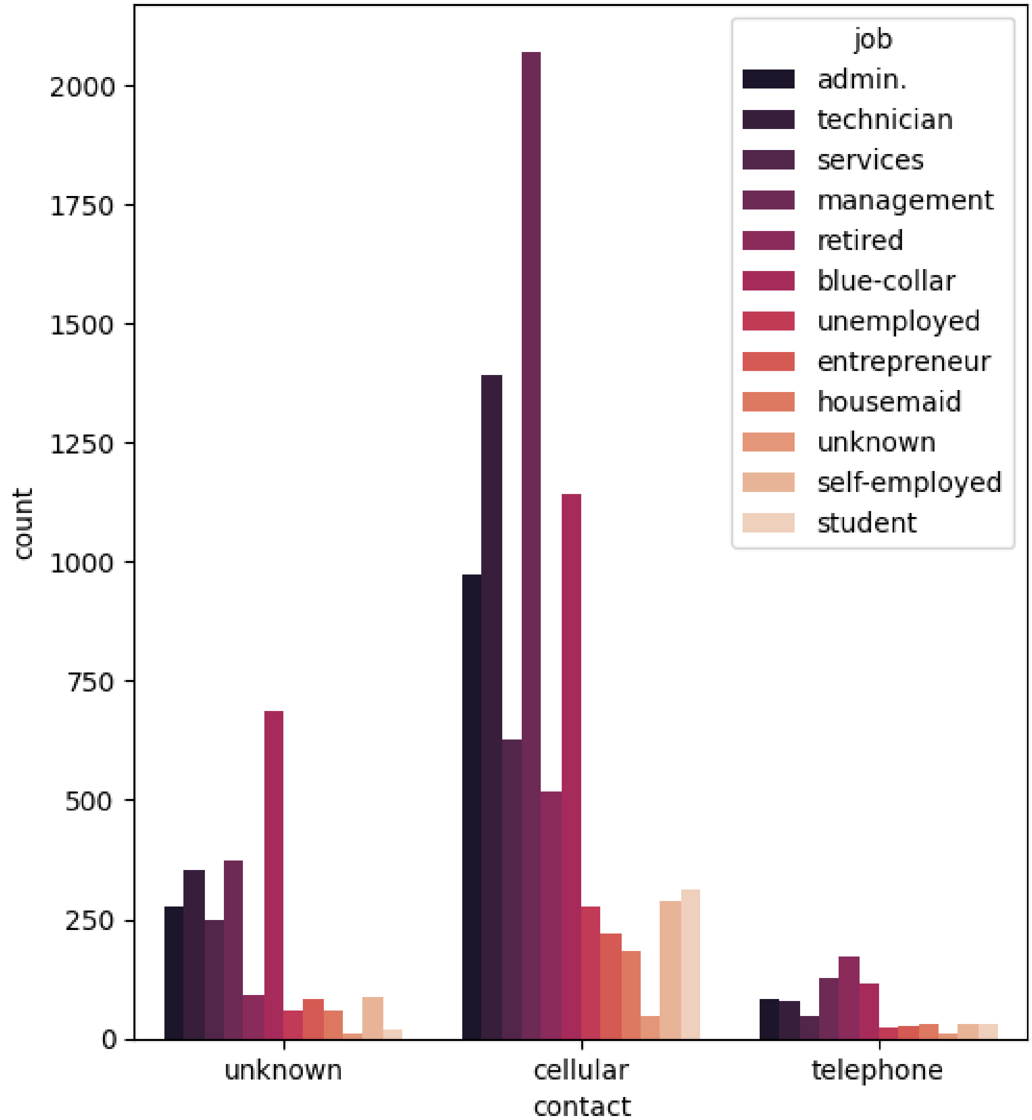


The distribution of jobs is almost the same across people who were married, single and divorced but the number of conversion in single and divorced in higher than married people.

[ ]:
```
df.groupby(["default","job"]).deposit.value_counts().unstack()
```

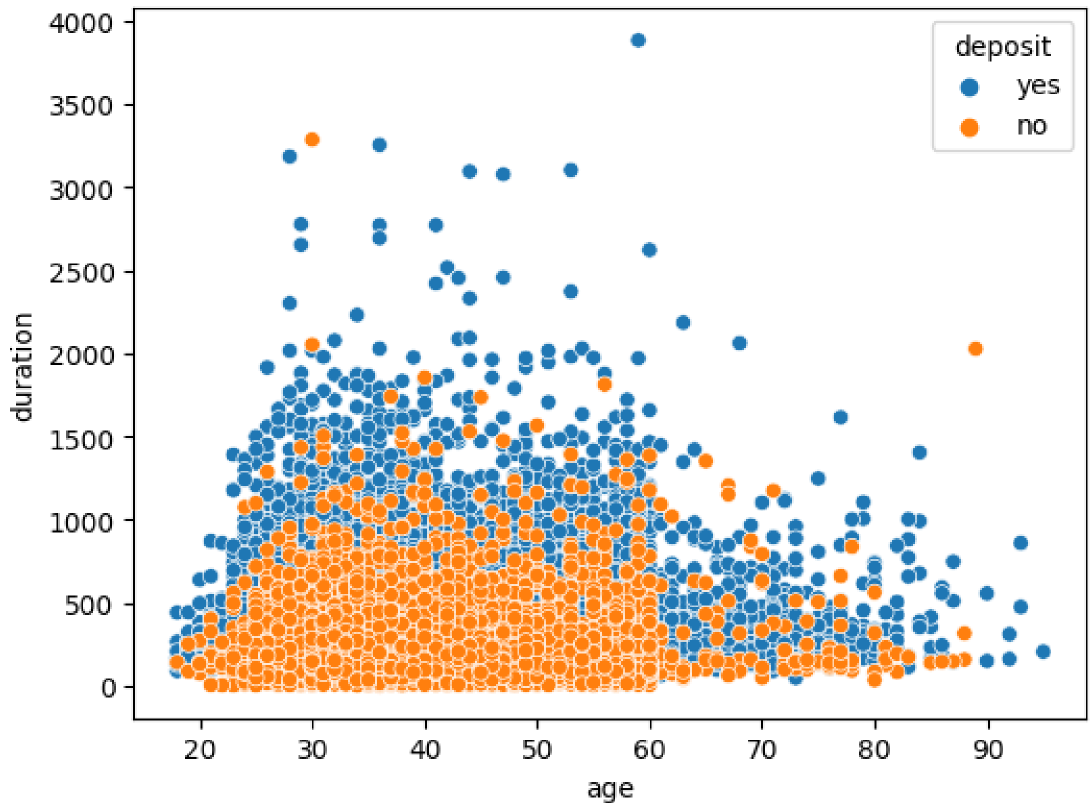
[ ]:
```
deposit                  no     yes
default job
no      admin.         695.0   628.0
        blue-collar   1210.0   693.0
        entrepreneur   201.0   117.0
        housemaid      158.0   108.0
        management    1234.0  1293.0
        retired        258.0   515.0
        self-employed  212.0   185.0
        services       551.0   365.0
        student         90.0   269.0
        technician     964.0   830.0
        unemployed     149.0   200.0
        unknown         35.0    34.0
yes     admin.           8.0     3.0
        blue-collar     26.0    15.0
        entrepreneur     4.0     6.0
        housemaid        7.0     1.0
        management      31.0     8.0
        retired          4.0     1.0
        self-employed    6.0     2.0
        services         3.0     4.0
        student          1.0     NaN
        technician      19.0    10.0
        unemployed       6.0     2.0
        unknown          1.0     NaN
```

```
[ ]: plt.figure(figsize=(6,7))
     g=sns.countplot(x=df["contact"],hue="job",data=df,palette="rocket")
```




```
[ ]: sns.scatterplot(data=df, x="age", y="duration",hue= "deposit")
```

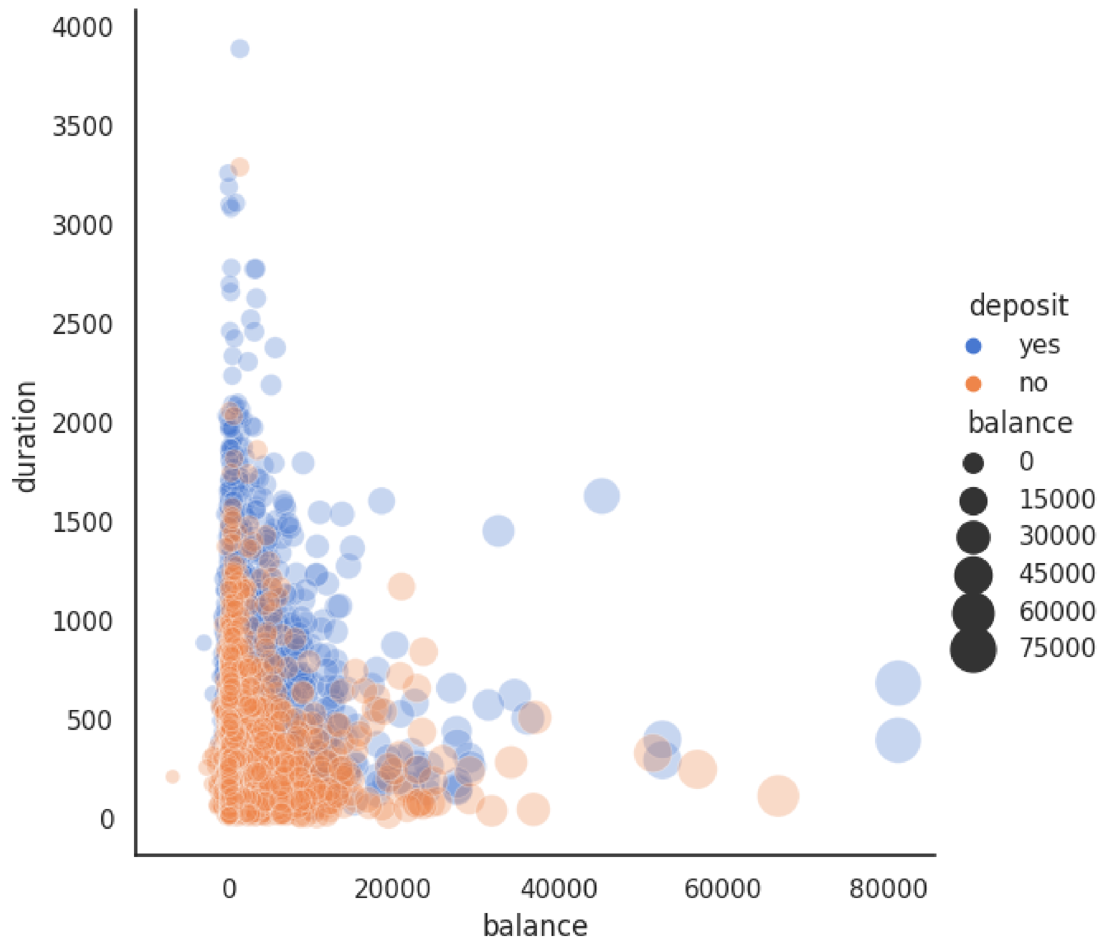
```
[ ]: <Axes: xlabel='age', ylabel='duration'>
```



- Duration is one of the most important feature in the data, as it does make sense that the more time you talk to an agent, the more you would be inclined towards opening a term deposit.
- you can also see that the number of conversions is more for age category above 60.

[ ]:
```
import seaborn as sns
sns.set_theme(style="white")

# Plot miles per gallon against horsepower with other semantics
sns.relplot(x="balance", y="duration", hue="deposit",size = "balance",
            sizes=(40, 400), alpha=.3, palette="muted",
            height=6, data=df)
```

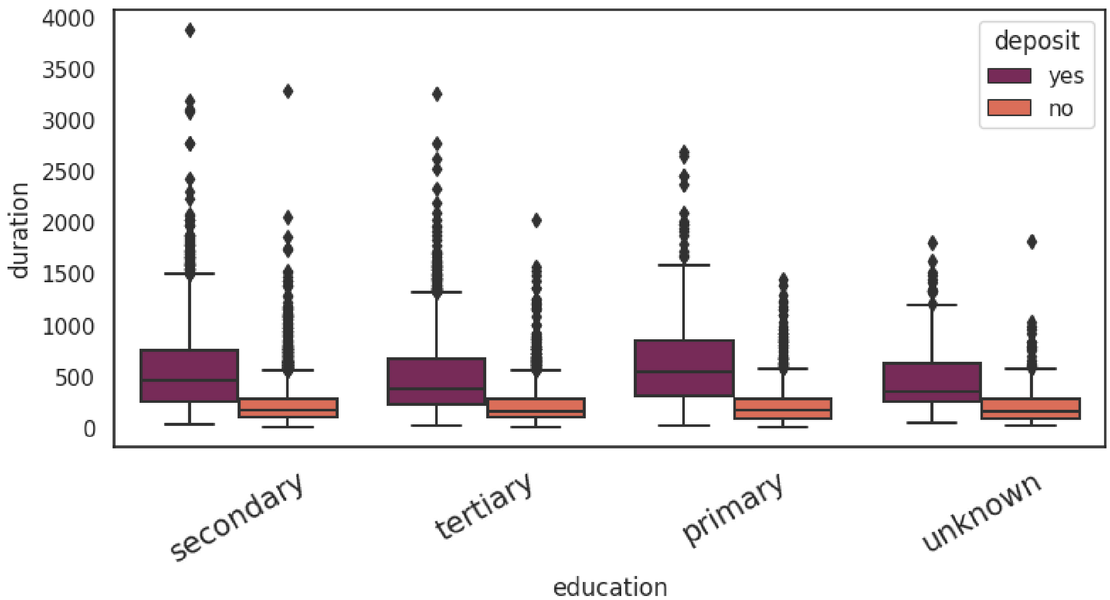
[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7d13083eff10>
```



- The ratio of Yes to NO as the balance increase is definitely more when it is in the range of 0-20000$.
- Higher the balance, more is the probability of opening a Term deposit.

[ ]:
```
plt.figure(figsize=(9,4))
g=sns.
 ↪boxplot(x=df["education"],y=df["duration"],hue=df["deposit"],palette="rocket",data=df)
g.set_xticklabels(g.get_xticklabels(),rotation=30,fontsize=15)
```

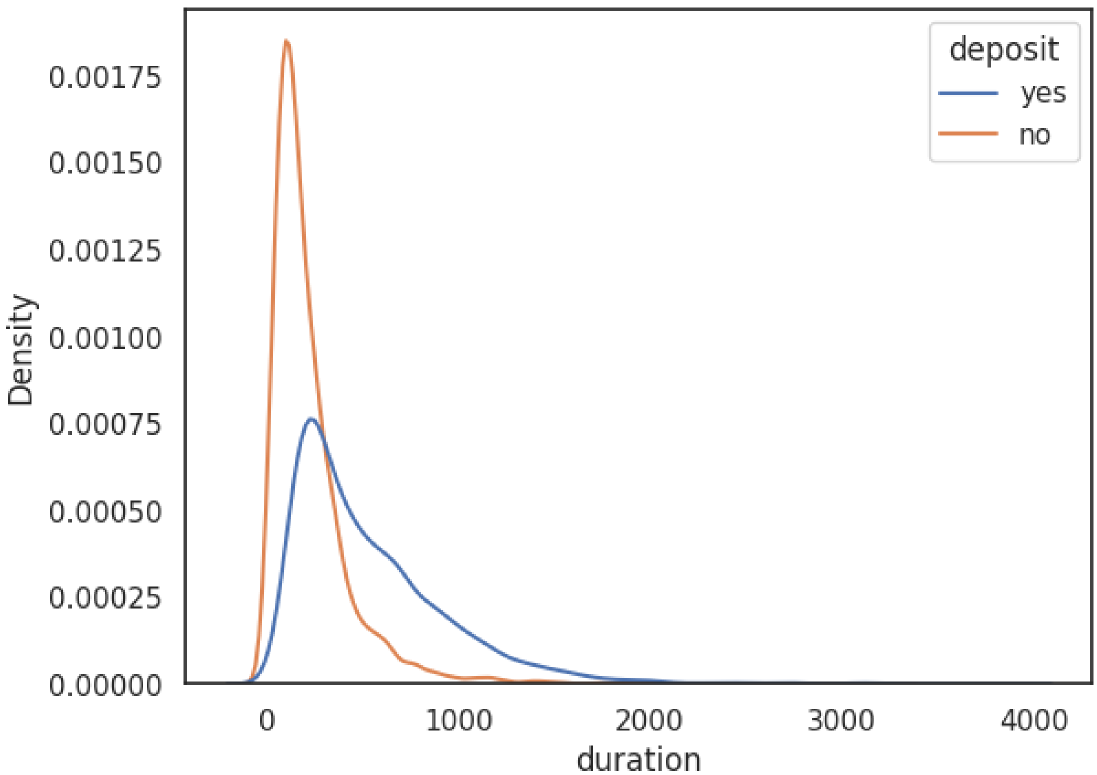
[ ]:
```
[Text(0, 0, 'secondary'),
 Text(1, 0, 'tertiary'),
 Text(2, 0, 'primary'),
 Text(3, 0, 'unknown')]
```



The duration of call in all the classes in education is considerably more when the customer ended up opening a term deposit.

[ ]:
```
sns.kdeplot(df,x = "duration",hue = "deposit")
```

[ ]:
```
<Axes: xlabel='duration', ylabel='Density'>
```



Finally, once can see how much the duration of the call influences in deciding if the customer ends up opening a term deposit. Hence, we have to make sure the test and train data is split equally to represenent the same distribution.

```
[ ]:
```

5gyfpyfp8

October 22, 2023

0.0.1 Question 3:

[26]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
from sklearn.ensemble import VotingClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
import warnings
warnings.filterwarnings('ignore')
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder,OrdinalEncoder,LabelEncoder
```

[3]:
```
data= pd.read_csv("./bank.csv")
df=data
```

[4]:
```
#Let's calculate the percentage of missing values in each column.
perc_missing = pd.DataFrame((df.isna().sum()/len(df)) * 100,columns =␣
↪["Perecentage Missing"])
perc_missing
#sns.heatmap(df.isnull(), cbar=False)
```

[4]:

	Perecentage Missing
age	0.0
job	0.0
marital	0.0
education	0.0
default	0.0
balance	0.0
housing	0.0

```
loan             0.0
contact          0.0
day              0.0
month            0.0
duration         0.0
campaign         0.0
pdays            0.0
previous         0.0
poutcome         0.0
deposit          0.0
```

[ ]:
```
corr_matrix = df.corr(method="pearson",numeric_only = True)
corr_matrix
```

[ ]:

	age	balance	day	duration	campaign	pdays	previous
age	1.000000	0.112300	-0.000762	0.000189	-0.005278	0.002774	0.020169
balance	0.112300	1.000000	0.010467	0.022436	-0.013894	0.017411	0.030805
day	-0.000762	0.010467	1.000000	-0.018511	0.137007	-0.077232	-0.058981
duration	0.000189	0.022436	-0.018511	1.000000	-0.041557	-0.027392	-0.026716
campaign	-0.005278	-0.013894	0.137007	-0.041557	1.000000	-0.102726	-0.049699
pdays	0.002774	0.017411	-0.077232	-0.027392	-0.102726	1.000000	0.507272
previous	0.020169	0.030805	-0.058981	-0.026716	-0.049699	0.507272	1.000000

Question 3 Answer:

- With stratified split, one can mention the variable from which equal portions should be taken for both test and train.
- Here i split train and test into 80/20.Then i split train data again into train and validation with 80 /20 split.

[143]:
```
#split training and testing:
train,test = train_test_split(df, test_size=0.2, stratify=df["deposit"],␣
↪random_state=42)
```

[144]:
```
train["deposit"].value_counts()/len(train)
```

[144]:
```
no     0.526151
yes    0.473849
Name: deposit, dtype: float64
```

[145]:
```
test["deposit"].value_counts()/len(test)
```

[145]:
```
no     0.526198
yes    0.473802
Name: deposit, dtype: float64
```

For validation set, gridsearch CV automatically splits the train and multiple cross validation data sets with 80/20 split.

0.0.2 Modelling

[146]:
```
#split into X and Y and encode the Target variable with categorical encoding
x_train,x_test =train.drop(columns=["deposit"]),test.drop(columns=["deposit"])
y_train,y_test = train["deposit"],test["deposit"]
encode = LabelEncoder()
y_train = encode.fit_transform(y_train)
y_test = encode.fit_transform(y_test)
```

[147]:
```
#check which category it encodes into:
encode.transform(["no"])
```

[147]:
```
array([0])
```

[148]:
```
#one hot encoding for nominal and label encoder for ordinal
binary_categorical_features =␣
↪["marital","default","loan","contact","job","month","poutcome","housing"]
ordinal_categorical_features = ["education"]
target_column = ["deposit"]
numeric_features =␣
↪["age","balance","previous","pdays","campaign","duration","day"]
```

Creating pre-processing pipeline

[149]:
```
columnTransformer = ColumnTransformer(
    transformers=[('bin_cat', OneHotEncoder(handle_unknown='ignore' ),␣
↪binary_categorical_features),
                  ('ord_cat',OrdinalEncoder(),ordinal_categorical_features),
     ('num', StandardScaler(), numeric_features)])

x_train = columnTransformer.fit_transform(x_train)
```

[150]:
```
x_train
```

[150]:
```
array([[ 0.        ,  1.        ,  0.        , …,  0.17848144,
        -0.34398786,  1.81720843],
       [ 0.        ,  1.        ,  0.        , …, -0.5400311 ,
        -0.3179716 , -1.50481988],
       [ 0.        ,  0.        ,  1.        , …, -0.5400311 ,
        -0.48563192, -0.55566894],
       …,
       [ 0.        ,  1.        ,  0.        , …, -0.5400311 ,
        -0.48852261, -1.74210762],
       [ 0.        ,  1.        ,  0.        , …, -0.5400311 ,
```

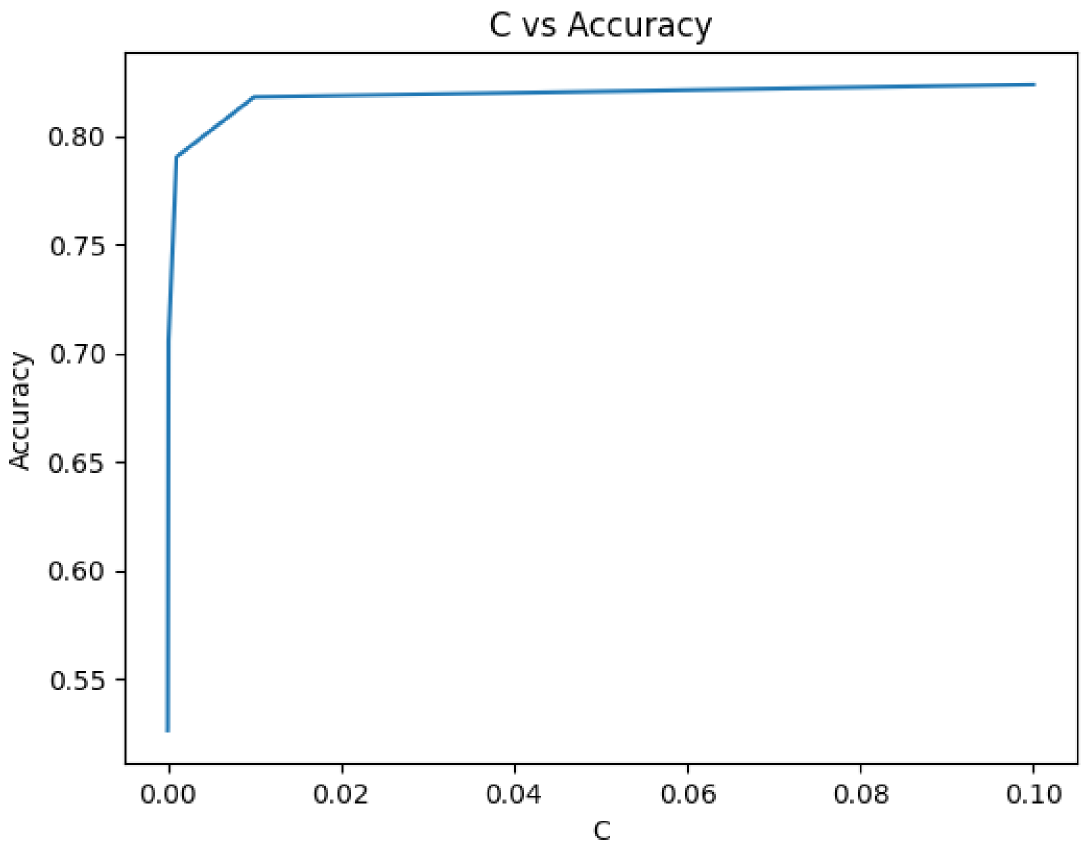
```
       0.54345553, -1.26753215],
     [ 0.        ,  1.        ,  0.        , …, -0.5400311 ,
      -0.71688752, -0.67431281]])
```

0.0.3 Question 4 A:

Softmax Regression for binomial class

- Hyperparameter = C

[78]:
```python
softmax_reg = LogisticRegression()
#automatic stratified split happens here
C = [0.000001, 0.00001, 0.0001,0.001,0.01, 0.1]
param_grid = {
    'C': C}
grid_search = GridSearchCV(softmax_reg, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=C, y=accuracy_scores)
# Add labels and a title
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title('C vs Accuracy')

# Show the plot
plt.show()
```



One can see that, when C is small[regularization parameter), the accuracy is less. Lesses the C, stronger the regularization. But as we increase C, the accuracy also increases, decreasing the amount of regularization,

[79]:
```
scores = pd.DataFrame(data =C,columns=["C"])
scores["accuracy_scores"] = accuracy_scores
scores
```

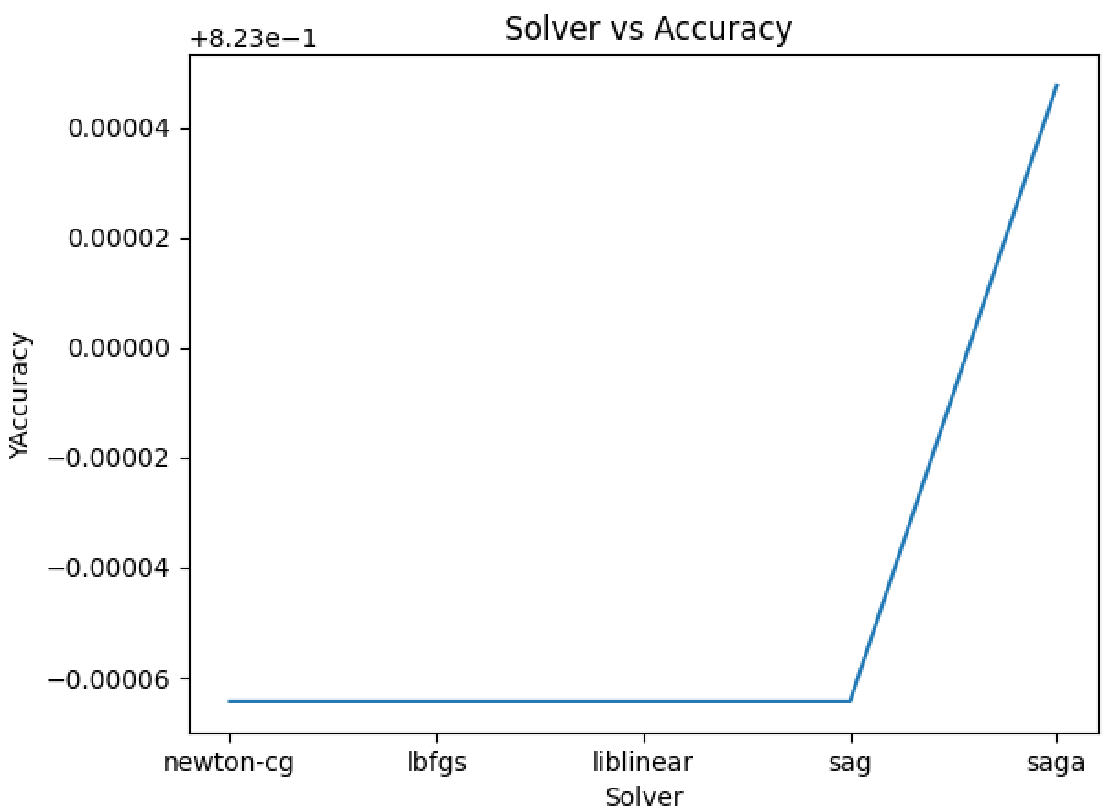
[79]:

	C	accuracy_scores
0	0.000001	0.526151
1	0.000010	0.527047
2	0.000100	0.705790
3	0.001000	0.790346
4	0.010000	0.818008
5	0.100000	0.823720

- Hyperparameter = Solver

[85]:
```
solver = ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
param_grid = {
```

```
    'solver': solver}
grid_search = GridSearchCV(softmax_reg, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=solver, y=accuracy_scores)
# Add labels and a title
plt.xlabel('Solver')
plt.ylabel('YAccuracy')
plt.title(' Solver vs Accuracy')
# Show the plot
plt.show()
```



[86]:
```
scores = pd.DataFrame(data =solver,columns=["Solver"])
scores["accuracy_scores"] = accuracy_scores
scores
```

[86]:

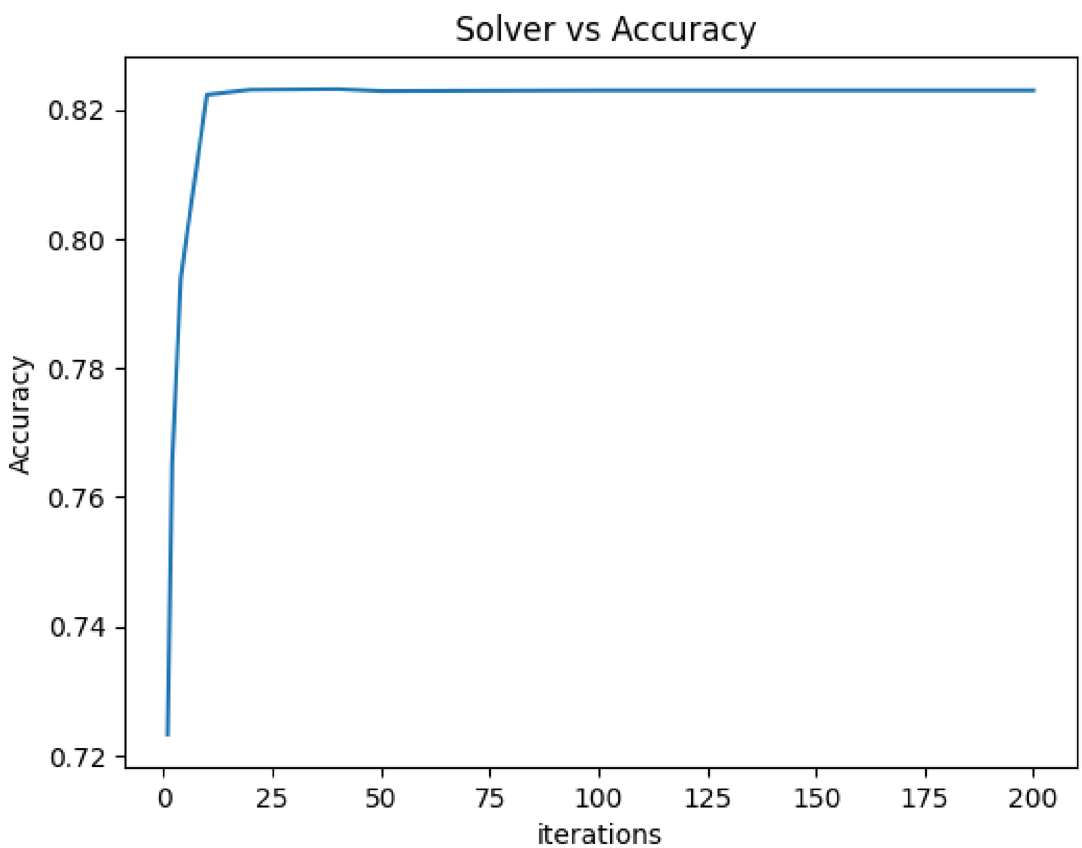
	Solver	accuracy_scores
0	newton-cg	0.822936
1	lbfgs	0.822936

```
2  liblinear         0.822936
3        sag         0.822936
4       saga         0.823048
```

Almost all the solvers have same accuracy scores apart from saga, which is effecient for large datasets[111,000 observations]. It also has varienty of penalty parameters such as l1,l2 and elastic net making it very versatile and suitable for binary and multiclass classification

- Max number of Iterations

[87]:
```
max_iter= [1,2,4,5,10,20,40,50,100,200]
param_grid = {
    'max_iter': max_iter}
grid_search = GridSearchCV(softmax_reg, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=max_iter, y=accuracy_scores)
# Add labels and a title
plt.xlabel('iterations')
plt.ylabel('Accuracy')
plt.title(' Solver vs Accuracy')

# Show the plot
plt.show()
```



[88]:

```
scores = pd.DataFrame(data = max_iter,columns=["max_iter"])
scores["accuracy_scores"] = accuracy_scores
scores
```

[88]:

	max_iter	accuracy_scores
0	1	0.723261
1	2	0.765371
2	4	0.793819
3	5	0.798746
4	10	0.822264
5	20	0.823048
6	40	0.823160
7	50	0.822824
8	100	0.822936
9	200	0.822936

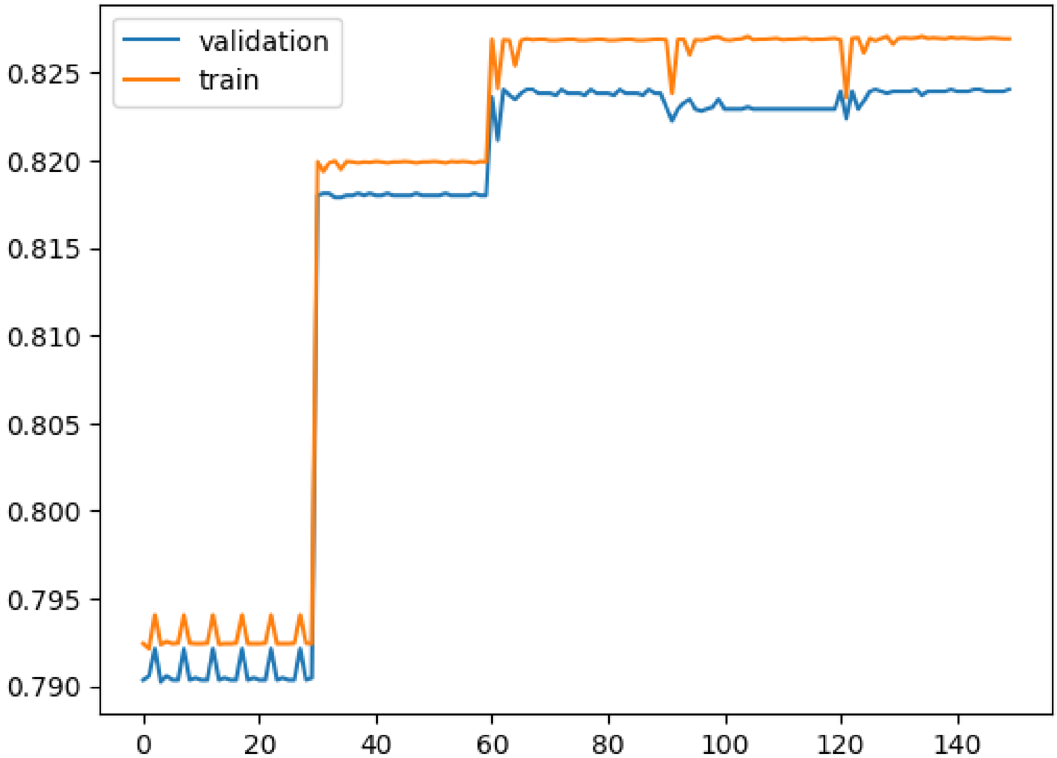
increasing the max_iter increases the accuracy scores. But after a particular range it becomes constant. Logistic regression algorithms are trained using optimization Algorithms such as gradient descent where the model converges to the local minima after certain number of iterations. hence providing with ebough iteration is important.

- Find the best hyperparameters for Logistic Regression

[89]:
```
#find best model
param_grid = {
    'C': [ 0.001, 0.01,0.1, 1.0, 10],
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
    'max_iter': [10,50,100,200, 300, 500]
}
grid_search = GridSearchCV(softmax_reg, param_grid=param_grid,
                           cv=5, n_jobs=-1,␣
 ↪scoring="accuracy",return_train_score=True)
grid_search.fit(x_train, y_train)
print("\ntuned hpyerparameters :(best parameters) ",grid_search.best_params_)
print("\naccuracy :",grid_search.best_score_)
```

```
tuned hpyerparameters :(best parameters)  {'C': 0.1, 'max_iter': 10, 'solver':
'liblinear'}

accuracy : 0.8240559471268911
```

[90]:
```
#plot graph
validation_scores = grid_search.cv_results_['mean_test_score']
train_scores = grid_search.cv_results_['mean_train_score']
plt.plot(validation_scores, label='validation')
plt.plot(train_scores, label='train')
plt.legend(loc='best')
plt.show()
```



- fit the model with best parameters

[93]:
```python
from sklearn.metrics import classification_report, confusion_matrix
best_log_model = grid_search.best_estimator_
y_pred = best_log_model.predict(columnTransformer.fit_transform(x_test))
# Compute precision, recall, and F1 score
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
# Print the results
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print(classification_report(y_test, y_pred))
```

```
Precision: 0.8338278931750742
Recall: 0.7967863894139886
F1 Score: 0.8148864185596907
              precision    recall  f1-score   support

           0       0.82      0.86      0.84      1175
           1       0.83      0.80      0.81      1058
```

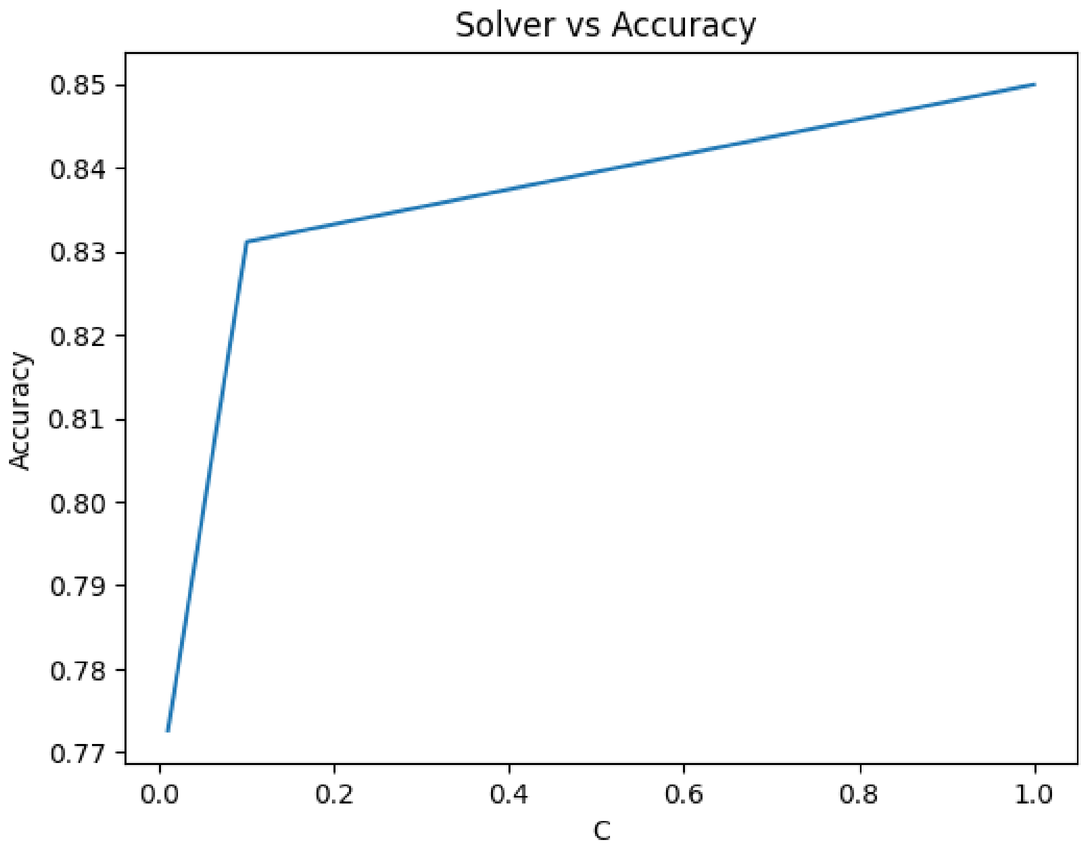
```
    accuracy                           0.83      2233
   macro avg       0.83      0.83      0.83      2233
weighted avg       0.83      0.83      0.83      2233
```

0.0.4 4b: SVMs

[94]:
```
#SVMs
from sklearn.svm import SVC
svm = SVC(probability=True)
```

- Hyperparameter : C

[95]:
```
C=[0.01,0.1,1.0]
param_grid = {
    'C': C}
grid_search = GridSearchCV(svm, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=C, y=accuracy_scores)
# Add labels and a title
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title(' Solver vs Accuracy')
# Show the plot
plt.show()
```



One can see that, when C is small[regularization parameter), the accuracy is less. Lesses the C, stronger the regularization. But as we increase C, the accuracy also increases, decreasing the amount of regularization.

[75]:
```
scores = pd.DataFrame(data = C,columns=["C"])
scores["accuracy_scores"] = accuracy_scores
scores
```

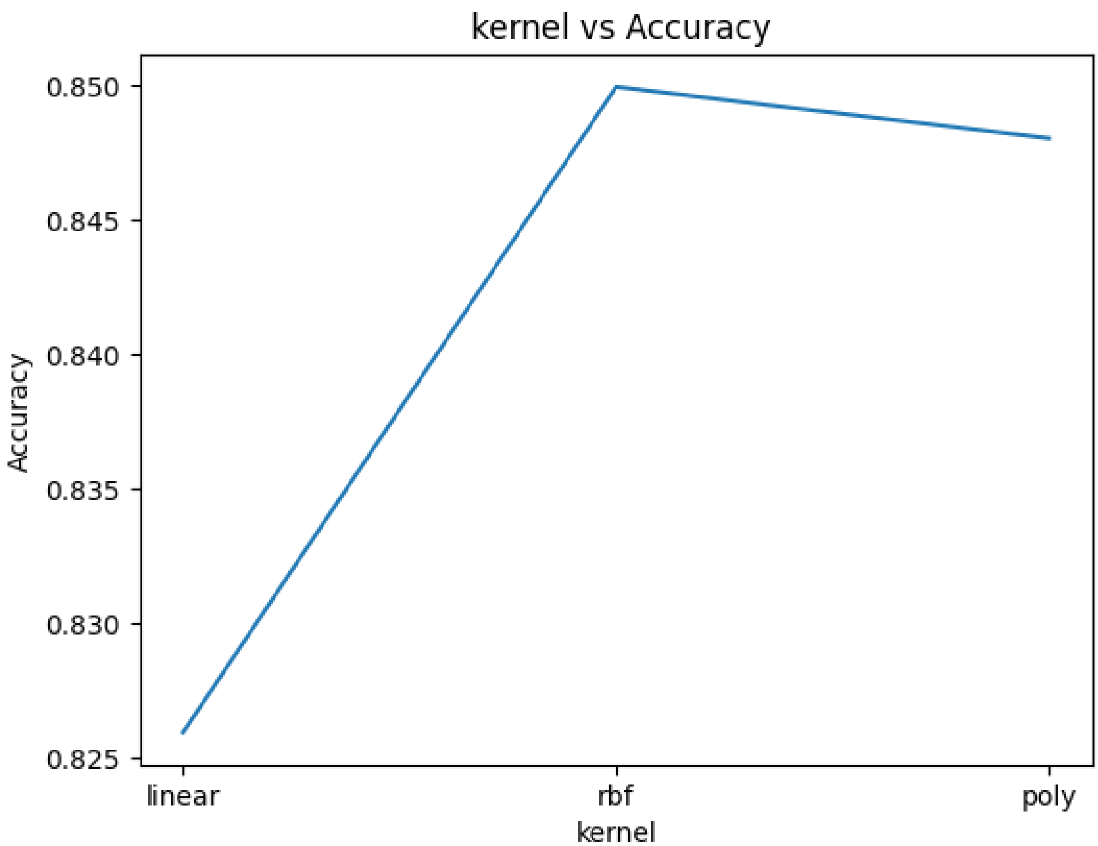
[75]:

	C	accuracy_scores
0	0.01	0.772651
1	0.10	0.831112
2	1.00	0.849927

- Hyperparameter: Kernel

[76]:
```
kernel=['linear', 'rbf', 'poly']
param_grid = {
    'kernel': kernel}
grid_search = GridSearchCV(svm, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
```

```python
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=kernel, y=accuracy_scores)
# Add labels and a title
plt.xlabel('kernel')
plt.ylabel('Accuracy')
plt.title(' kernel vs Accuracy')
# Show the plot
plt.show()
```



[96]:

```python
scores = pd.DataFrame(data = kernel,columns=["Kernel"])
scores["accuracy_scores"] = accuracy_scores
scores
```

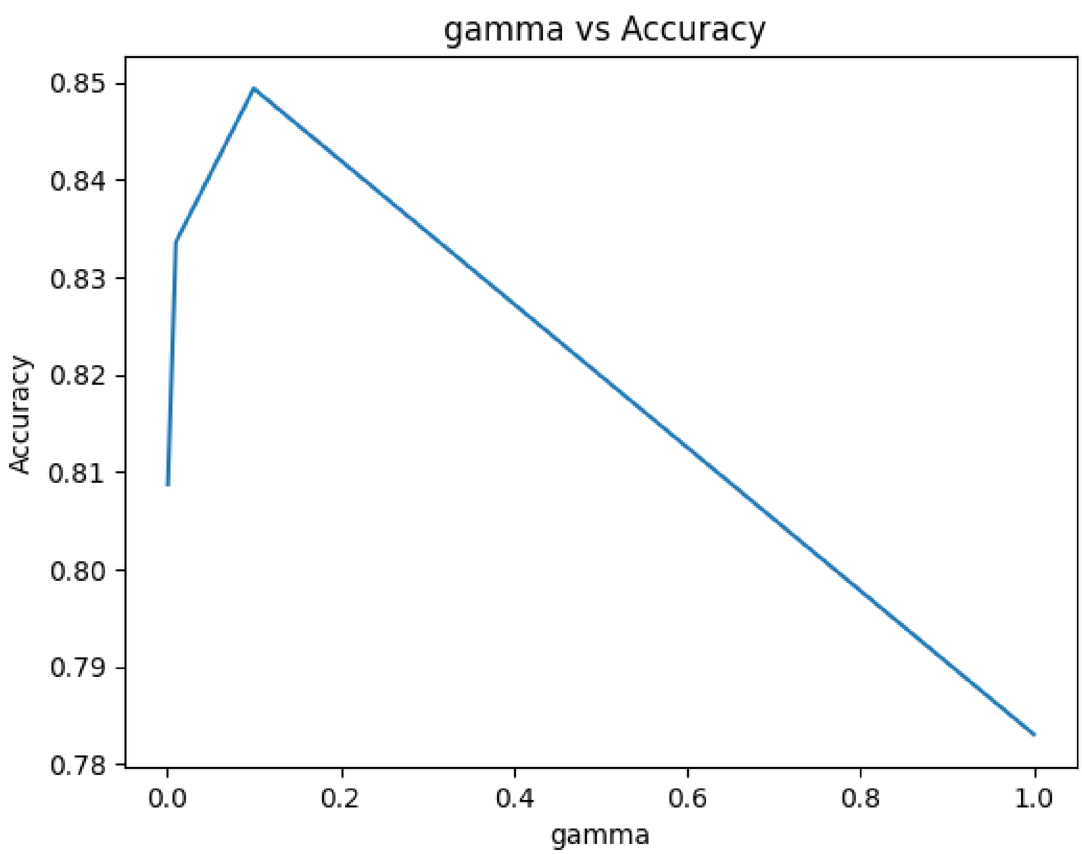
[96]:

	Kernel	accuracy_scores
0	linear	0.772651
1	rbf	0.831112
2	poly	0.849927

Ine can see that rbf kernel has higher accuracy compared to other kernels.Because the use case in hand is complex and may have decision boundaries that are complex, rbf kernel with its ability to create non linear transformation has performed better.

- Hyperparameter Gamma

[97]:
```
gamma= [0.001, 0.01, 0.1, 1]
param_grid = {
    'gamma': gamma}
grid_search = GridSearchCV(svm, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=gamma, y=accuracy_scores)
# Add labels and a title
plt.xlabel('gamma')
plt.ylabel('Accuracy')
plt.title(' gamma vs Accuracy')
# Show the plot
plt.show()
```



[98]:
```
scores = pd.DataFrame(data = gamma,columns=["Gamma"])
scores["accuracy_scores"] = accuracy_scores
scores
```

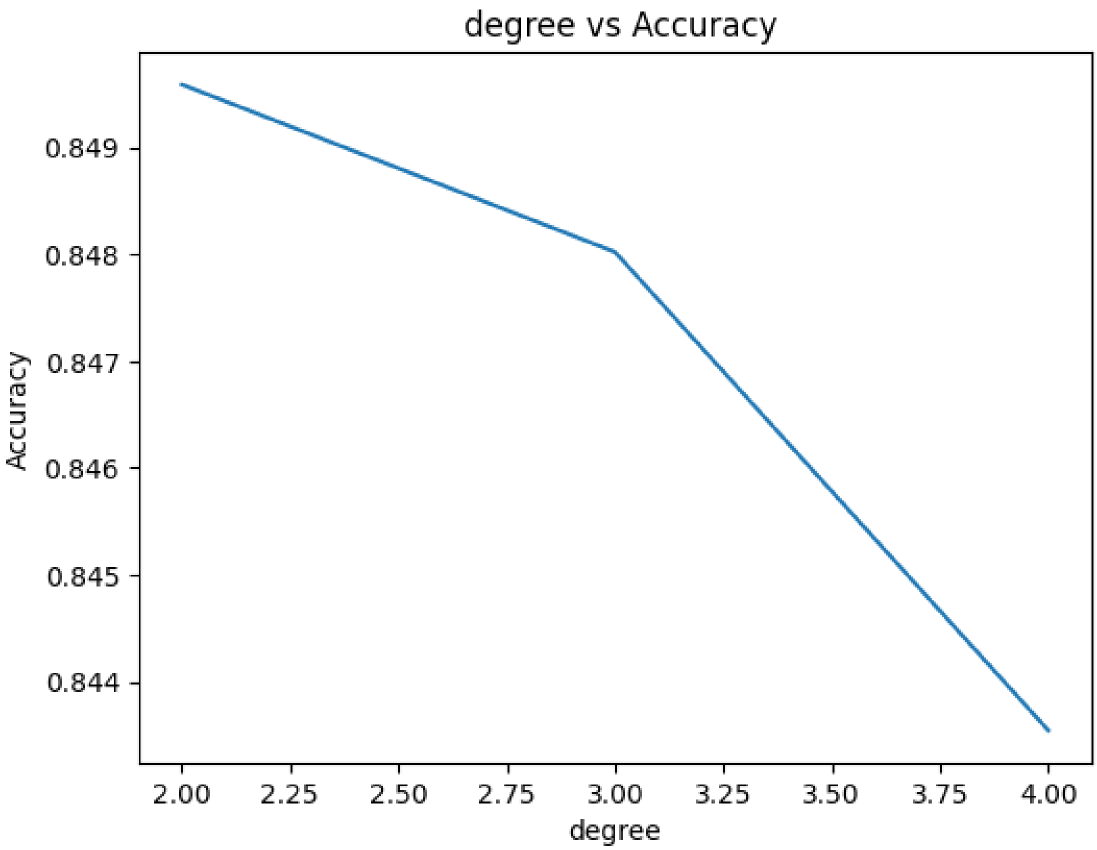
[98]:

	Gamma	accuracy_scores
0	0.001	0.808713
1	0.010	0.833576
2	0.100	0.849367
3	1.000	0.783066

gamma parameter defines how smooth and generalized athe decsion bounday has to be. Higher the gamma, closely it fits to the training samples leading to overfitting which in turn decrease accuracy.

- Hyperparameter : Polynomial degree

[99]:
```
degree= [2, 3, 4]
param_grid = {
    'degree': degree}
grid_search = GridSearchCV(SVC(kernel="poly"), param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=degree, y=accuracy_scores)
# Add labels and a title
plt.xlabel('degree')
plt.ylabel('Accuracy')
plt.title(' degree vs Accuracy')
# Show the plot
plt.show()
```



[100]:
```
scores = pd.DataFrame(data = degree,columns=["degree"])
scores["accuracy_scores"] = accuracy_scores
scores
```

[100]:

	degree	accuracy_scores
0	2	0.849591
1	3	0.848022
2	4	0.843543

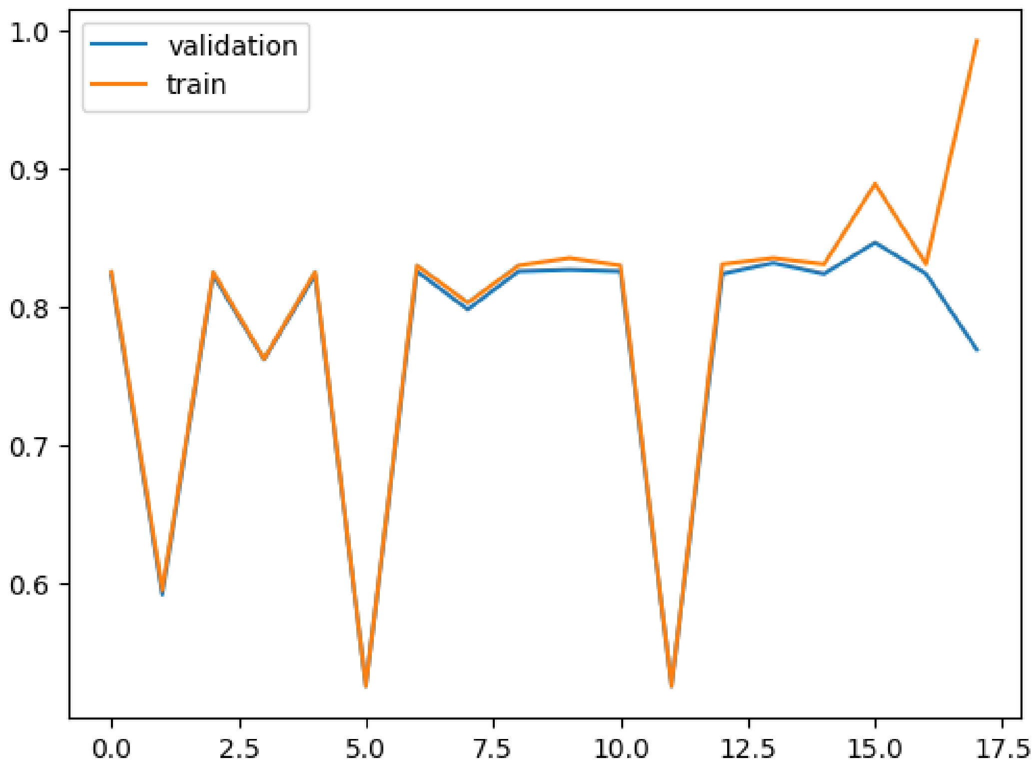
a lower polynomial degree is used when the variables are linearly separable.Increasing the number of degress leads to model being overft, which inturn reduces the testing accuracy.

- find and fit the best model to SVM

[156]:
```
#best model
param_grid = {
    'C': [0.01,0.1, 1],
    'kernel': ['linear', 'rbf'],
    'gamma': [0.01, 0.1, 1]
}
grid_search = GridSearchCV(svc, param_grid=param_grid,
```

```
                              cv=2, n_jobs=-1,␣
  ↪scoring="accuracy",return_train_score=True)
grid_search.fit(x_train, y_train)
print("\n")
print("tuned hpyerparameters :(best parameters) ",grid_search.best_params_)
print("accuracy :",grid_search.best_score_)
validation_scores = grid_search.cv_results_['mean_test_score']
train_scores = grid_search.cv_results_['mean_train_score']
plt.plot(validation_scores, label='validation')
plt.plot(train_scores, label='train')
plt.legend(loc='best')
plt.show()
```

```
tuned hpyerparameters :(best parameters)  {'C': 1, 'gamma': 0.1, 'kernel':
'rbf'}
accuracy : 0.8465672374140567
```



- print performance metrics for best model

```
[102]: from sklearn.metrics import classification_report, confusion_matrix
best_svm_model = grid_search.best_estimator_
y_pred = best_svm_model.predict(columnTransformer.fit_transform(x_test))
# Compute precision, recall, and F1 score
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
# Print the results
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print(classification_report(y_test, y_pred))
```

```
Precision: 0.8269402319357716
Recall: 0.8761814744801513
F1 Score: 0.8508490133088572
              precision    recall  f1-score   support

           0       0.88      0.83      0.86      1175
           1       0.83      0.88      0.85      1058

    accuracy                           0.85      2233
   macro avg       0.85      0.86      0.85      2233
weighted avg       0.86      0.85      0.85      2233
```

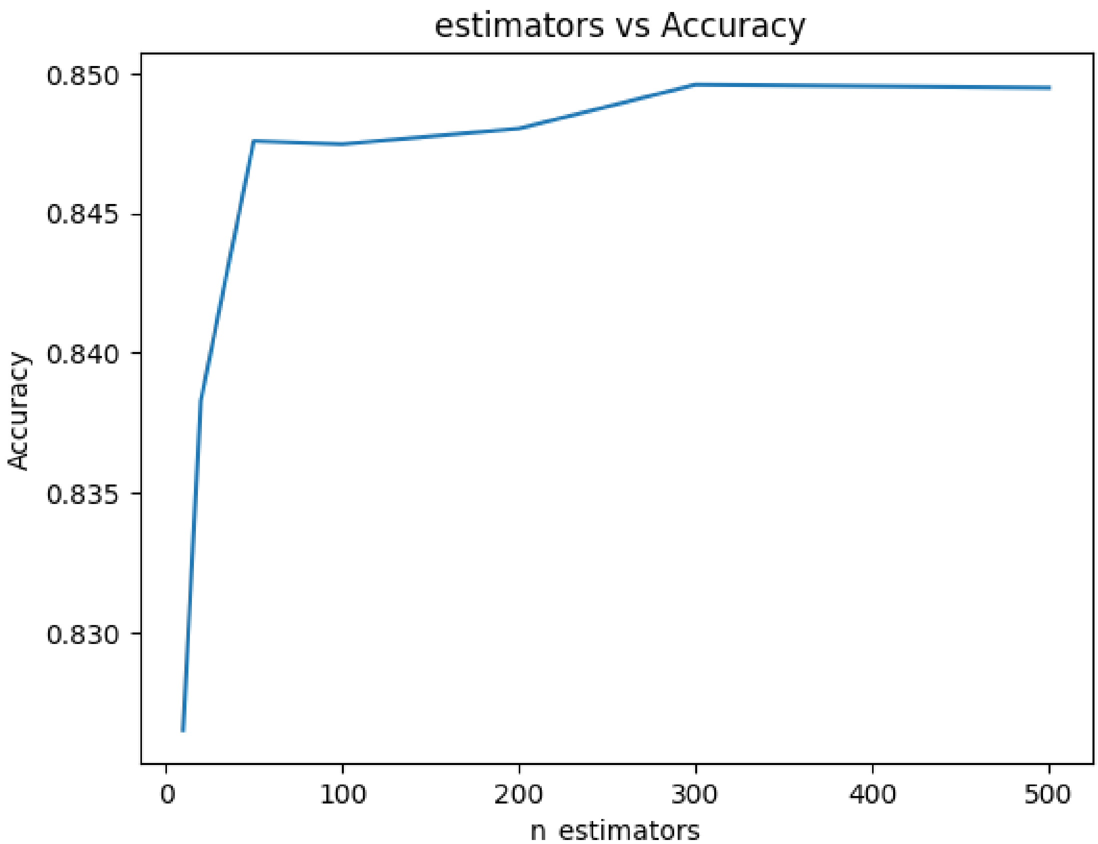
0.0.5 4c Random Forest Classifier

```
[103]: #random forest
from sklearn.ensemble import RandomForestClassifier
```

```
[104]: rf_clf = RandomForestClassifier()
```

```
[105]: n_estimators=[10,20,50, 100, 200, 300, 500]
param_grid = {
    'n_estimators': n_estimators}
grid_search = GridSearchCV(rf_clf, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=n_estimators, y=accuracy_scores)
# Add labels and a title
plt.xlabel('n_estimators')
plt.ylabel('Accuracy')
plt.title(' estimators vs Accuracy')
# Show the plot
```

```
plt.show()
```



[106]:
```
scores = pd.DataFrame(data = n_estimators,columns=["n_estimators"])
scores["accuracy_scores"] = accuracy_scores
scores
```

[106]:

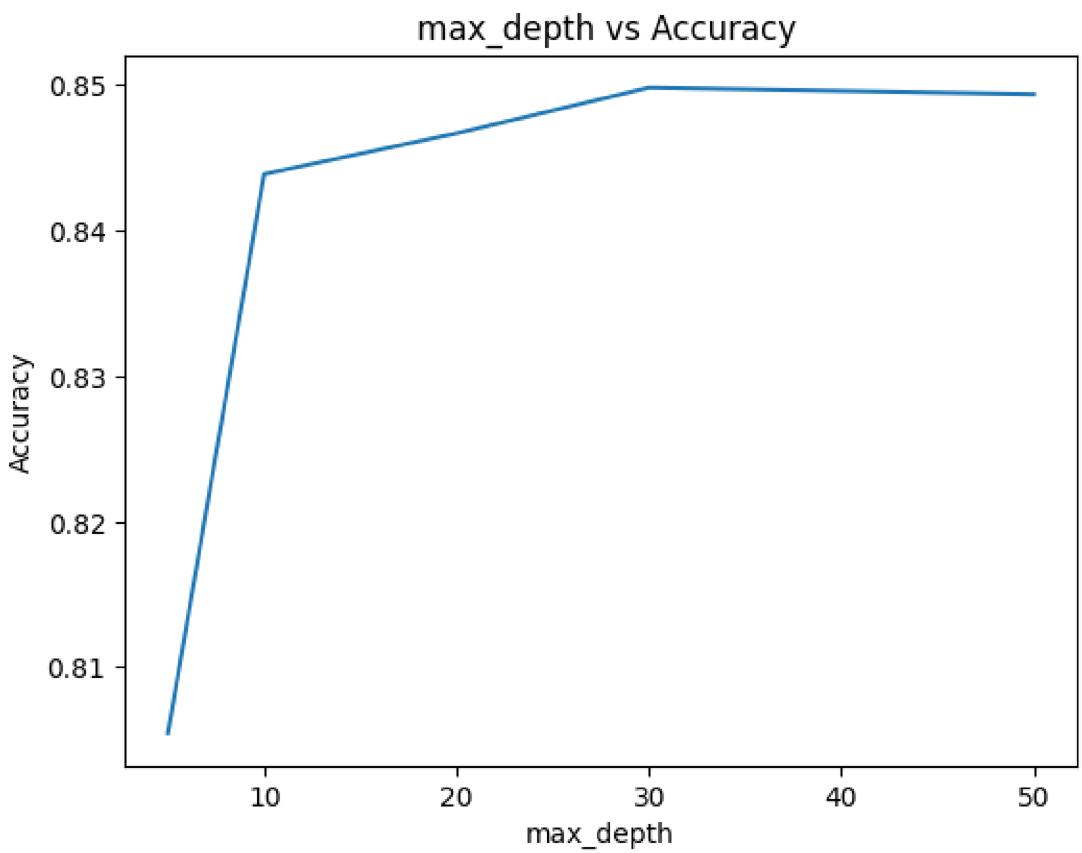
	n_estimators	accuracy_scores
0	10	0.826520
1	20	0.838279
2	50	0.847575
3	100	0.847463
4	200	0.848023
5	300	0.849591
6	500	0.849479

n_estimators defines the number of decision trees that needs to be created. higher the decsion tree, the probability of overfitting increases, and lower it is, the model might be underfit. In this scenario, one can see that the accuracy starts to decrease when estimators increase from 300-500, which means the model is overfitting.

- Hyper-parameter : max_depth

[107]:

```
max_depth= [0, 5, 10, 20, 30,50]
param_grid = {
    'max_depth': max_depth}
grid_search = GridSearchCV(rf_clf, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=max_depth, y=accuracy_scores)
# Add labels and a title
plt.xlabel('max_depth')
plt.ylabel('Accuracy')
plt.title(' max_depth vs Accuracy')
# Show the plot
plt.show()
```



[108]:

```
scores = pd.DataFrame(data = max_depth,columns=["max_depth"])
scores["accuracy_scores"] = accuracy_scores
scores
```

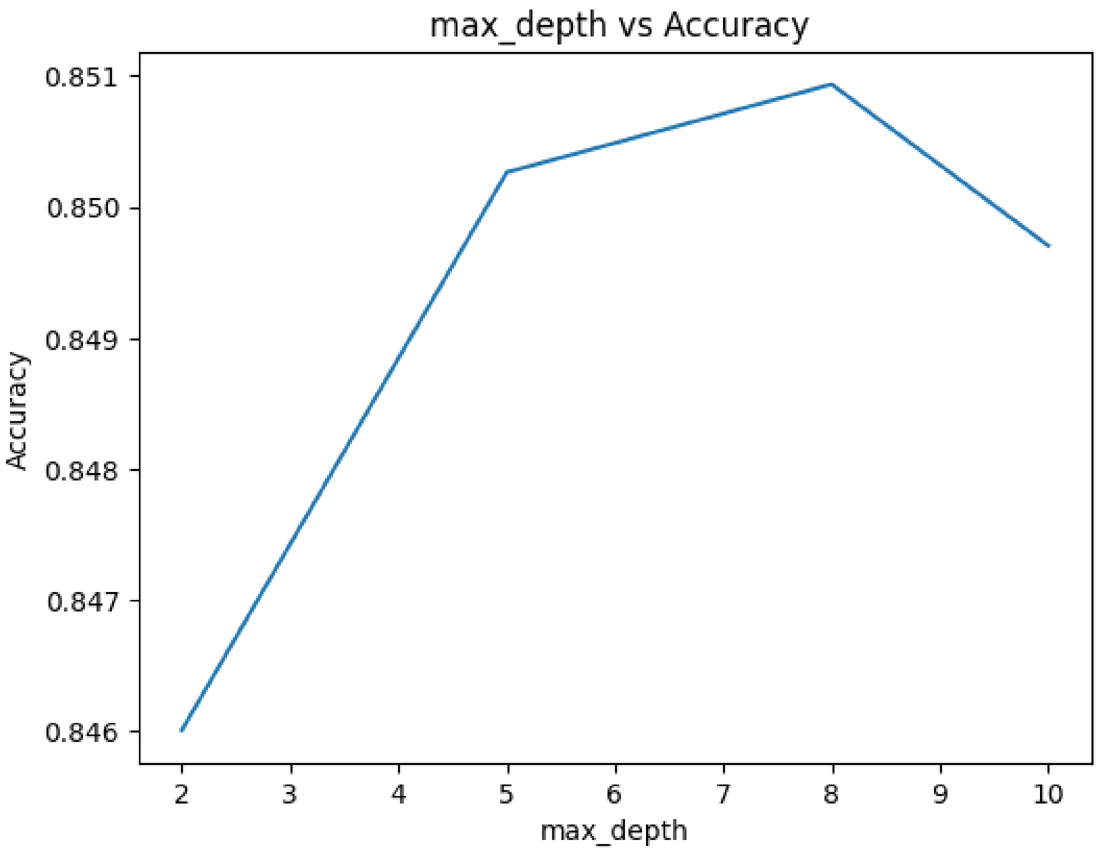
[108]:

	max_depth	accuracy_scores
0	0	NaN
1	5	0.805465
2	10	0.843879
3	20	0.846679
4	30	0.849815
5	50	0.849367

Lesser max_depth results in trees that are simpler and less complex. But as you increase the max_depth, the accuracy increases and starts decreasing after reaching the highest accuracy as the model will start to overfit.

[109]:

```
min_samples_split= [1,2, 5, 8,10]
param_grid = {
    'min_samples_split': min_samples_split}
grid_search = GridSearchCV(rf_clf, param_grid=param_grid,
                           cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=min_samples_split, y=accuracy_scores)
# Add labels and a title
plt.xlabel('max_depth')
plt.ylabel('Accuracy')
plt.title(' max_depth vs Accuracy')
# Show the plot
plt.show()
```



[110]:
```
scores = pd.DataFrame(data = min_samples_split,columns=["min_samples_split"])
scores["accuracy_scores"] = accuracy_scores
scores
```

[110]:

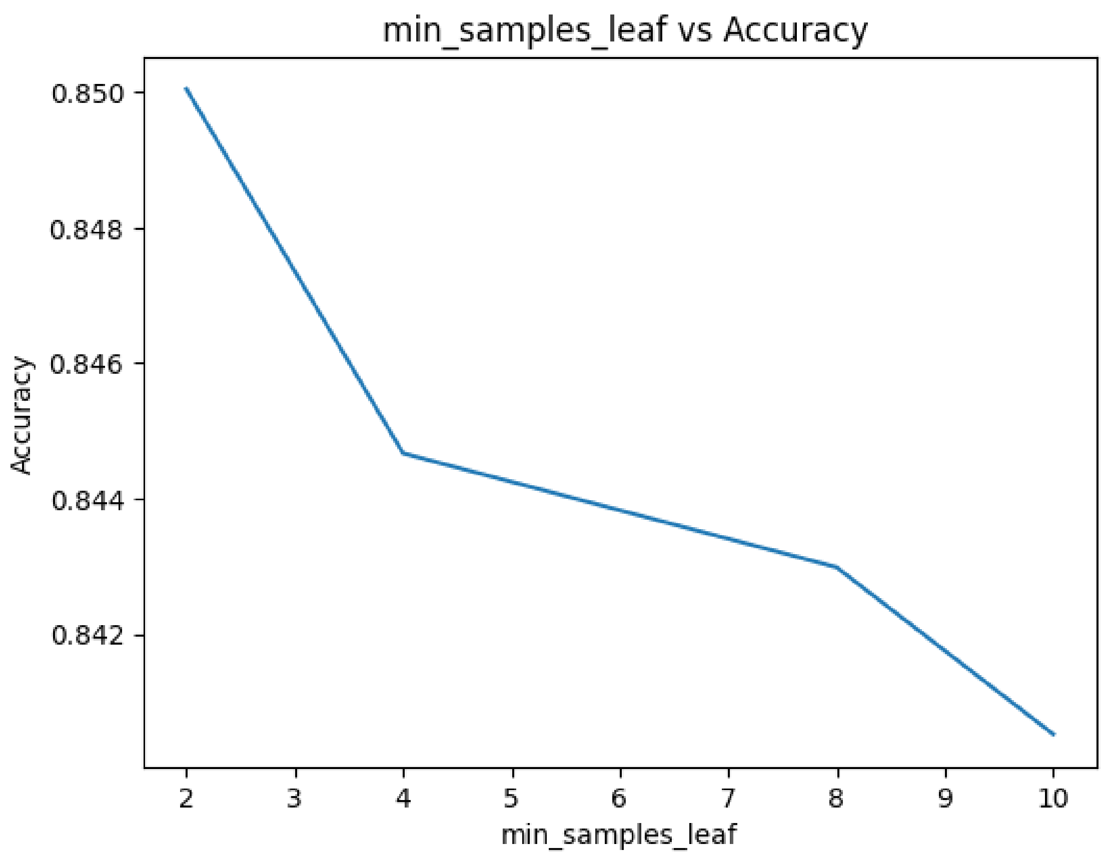
	min_samples_split	accuracy_scores
0	1	NaN
1	2	0.846006
2	5	0.850263
3	8	0.850935
4	10	0.849703

A samll "min samples split" means it allows the nodes to split even when there are less samples at node, which may lead to overfitting.

- hyperparameter = min_samples_leaf

[111]:
```
min_samples_leaf= [2,4,8,10]  # Minimum number of samples required to be at a␣
↪leaf node
param_grid = {
    'min_samples_leaf': min_samples_leaf}
grid_search = GridSearchCV(rf_clf, param_grid=param_grid,
```

```
                                    cv=5, n_jobs=-1, scoring="accuracy")
grid_search.fit(x_train, y_train)
results = grid_search.cv_results_
accuracy_scores = results["mean_test_score"]
sns.lineplot(x=min_samples_leaf, y=accuracy_scores)
# Add labels and a title
plt.xlabel('min_samples_leaf')
plt.ylabel('Accuracy')
plt.title(' min_samples_leaf vs Accuracy')
# Show the plot
plt.show()
```



[112]:
```
scores = pd.DataFrame(data = min_samples_leaf,columns=["min_samples_leaf"])
scores["accuracy_scores"] = accuracy_scores
scores
```

[112]:

	min_samples_leaf	accuracy_scores
0	2	0.850038
1	4	0.844663
2	8	0.842983

```
3                  10          0.840519
```

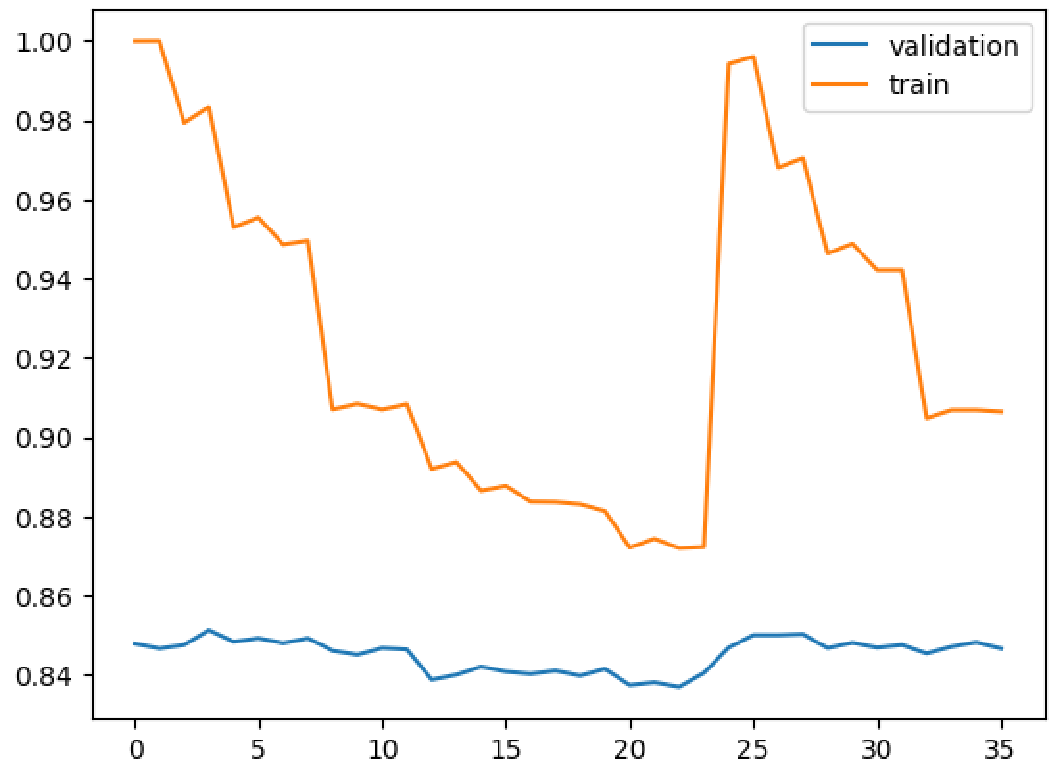
increasing the min_samples_leaf makes the model underfit the data as the trees becomes shallower and simpler, inturn reducing the overall accuracy.

- find the best parameter

[113]:
```
param_grid = {
    'n_estimators': [50, 100],  # Number of trees in the forest
    'max_depth': [None, 10, 20],  # Maximum depth of each tree
    'min_samples_split': [2, 5],  # Minimum number of samples required to split␣
 ↪an internal node
    'min_samples_leaf': [1, 2, 4],  # Minimum number of samples required to be␣
 ↪at a leaf node
}
```

[114]:
```
grid_search = GridSearchCV(rf_clf, param_grid=param_grid,
                           cv=5, n_jobs=-1,␣
 ↪scoring="accuracy",return_train_score=True)
grid_search.fit(x_train, y_train)
print("tuned hpyerparameters :(best parameters) ",grid_search.best_params_)
print("accuracy :",grid_search.best_score_)

validation_scores = grid_search.cv_results_['mean_test_score']
train_scores = grid_search.cv_results_['mean_train_score']
plt.plot(validation_scores, label='validation')
plt.plot(train_scores, label='train')
plt.legend(loc='best')
plt.show()
```

```
tuned hpyerparameters :(best parameters)  {'max_depth': None,
'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}
accuracy : 0.8512705418113493
```



- fit the best set of hyperparameters

[115]:
```
best_rf_clf_model = grid_search.best_estimator_
y_pred = best_rf_clf_model.predict(columnTransformer.fit_transform(x_test))
# Compute precision, recall, and F1 score
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
# Print the results
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print(classification_report(y_test, y_pred))
```

```
Precision: 0.8252853380158033
Recall: 0.888468809073724
F1 Score: 0.8557123350022757
              precision    recall  f1-score   support

           0       0.89      0.83      0.86      1175
           1       0.83      0.89      0.86      1058

    accuracy                           0.86      2233
```

```
   macro avg       0.86      0.86      0.86      2233
weighted avg       0.86      0.86      0.86      2233
```

Question 5: Ensemble Techniques

i.Voting Classifier

[158]:
```
#voting classifer
from sklearn.ensemble import VotingClassifier,AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
def Voting_classifier(method):
  ensemble_classifier = VotingClassifier(estimators =␣
 ↪[("log",best_log_model),("rf",best_rf_clf_model)],voting="soft")
  ensemble_classifier.fit(x_train,y_train)
  ensemble_predictions = ensemble_classifier.predict(columnTransformer.
 ↪fit_transform(x_test))
  # Evaluate the ensemble's performance
  ensemble_accuracy = accuracy_score(y_test, ensemble_predictions)
  print(method,"Voting classifier Results")
  print("Ensemble Test Accuracy for Voting Classifier:", ensemble_accuracy)
  for name, clf in ensemble_classifier.named_estimators_.items():
    print(name, "=", clf.score(columnTransformer.fit_transform(x_test), y_test))
Voting_classifier("hard")
```

```
hard Voting classifier Results
Ensemble Test Accuracy for Voting Classifier: 0.8540080609046127
log = 0.8284818629646216
rf = 0.8616211374832065
```

[159]:
```
Voting_classifier("soft")
```

```
soft Voting classifier Results
Ensemble Test Accuracy for Voting Classifier: 0.8490819525302284
log = 0.8284818629646216
rf = 0.8584863412449619
```

ii. Adaboost

[161]:
```
ensemble_classifier = VotingClassifier(estimators =␣
 ↪[("log",best_log_model),("rf",best_rf_clf_model)],voting="soft")
adaBoostClassifier = AdaBoostClassifier(estimator =␣
 ↪ensemble_classifier,n_estimators=100,)
adaBoostClassifier.fit(x_train,y_train)
```

[161]:
```
AdaBoostClassifier(estimator=VotingClassifier(estimators=[('log',
LogisticRegression(C=0.1,
max_iter=10,
```

```
solver='liblinear')),
                                                          ('rf',
RandomForestClassifier(min_samples_split=5))],
                                                    voting='soft'),
                   n_estimators=100)
```

[163]:

```
ensemble_predictions = adaBoostClassifier.predict(columnTransformer.
 ↪fit_transform(x_test))
# Evaluate the ensemble's performance
ensemble_accuracy = accuracy_score(y_test, ensemble_predictions)
print("Ensemble Accuracy for Adaboost:", ensemble_accuracy)
```

```
Ensemble Accuracy for Adaboost: 0.8625167935512763
```

iii. Bagging with DecsionTreeClassifier

[166]:

```
#bagging
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(DecisionTreeClassifier(),␣
 ↪n_estimators=500,max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(x_train, y_train)
bagging_predictions = bag_clf.predict(columnTransformer.fit_transform(x_test))
# Evaluate the ensemble's performance
bagging_accuracy = accuracy_score(y_test, bagging_predictions)
print("Ensemble Accuracy for bagging:", bagging_accuracy)
```

```
Ensemble Accuracy for bagging: 0.8159426780116436
```

iv. Gradient Boosting

[168]:

```
#gradient boosting
from sklearn.ensemble import GradientBoostingClassifier
gb_classifier = GradientBoostingClassifier(n_estimators=500, learning_rate=0.1,␣
 ↪max_depth=3,n_iter_no_change=20 )
gb_classifier.fit(x_train, y_train)
gb_predictions = gb_classifier.predict(columnTransformer.fit_transform(x_test))
gb_accuracy = accuracy_score(y_test, gb_predictions)
print("Gradient Boosting Classifier Accuracy:", gb_accuracy)
```

```
Gradient Boosting Classifier Accuracy: 0.8598298253470668
```

v. Stacking Classifier

[169]:

```
from sklearn.ensemble import StackingClassifier
stacking_clf = StackingClassifier(estimators=[('lr', best_log_model),('rf',␣
 ↪best_rf_clf_model)],
                                   final_estimator=best_log_model,
                                   cv=10)
stacking_clf.fit(x_train, y_train)
```

[169]:
```
StackingClassifier(cv=10,
                   estimators=[('lr',
                                LogisticRegression(C=0.1, max_iter=10,
                                                   solver='liblinear')),
                               ('rf',
                                RandomForestClassifier(min_samples_split=5))],
                   final_estimator=LogisticRegression(C=0.1, max_iter=10,
                                                      solver='liblinear'))
```

[171]:
```
stacking_pred = stacking_clf.predict(columnTransformer.fit_transform(x_test))
stacking_accuracy = accuracy_score(y_test, stacking_pred)
print("Gradient Boosting Classifier Accuracy:", stacking_accuracy)
```

```
Gradient Boosting Classifier Accuracy: 0.8557993730407524
```

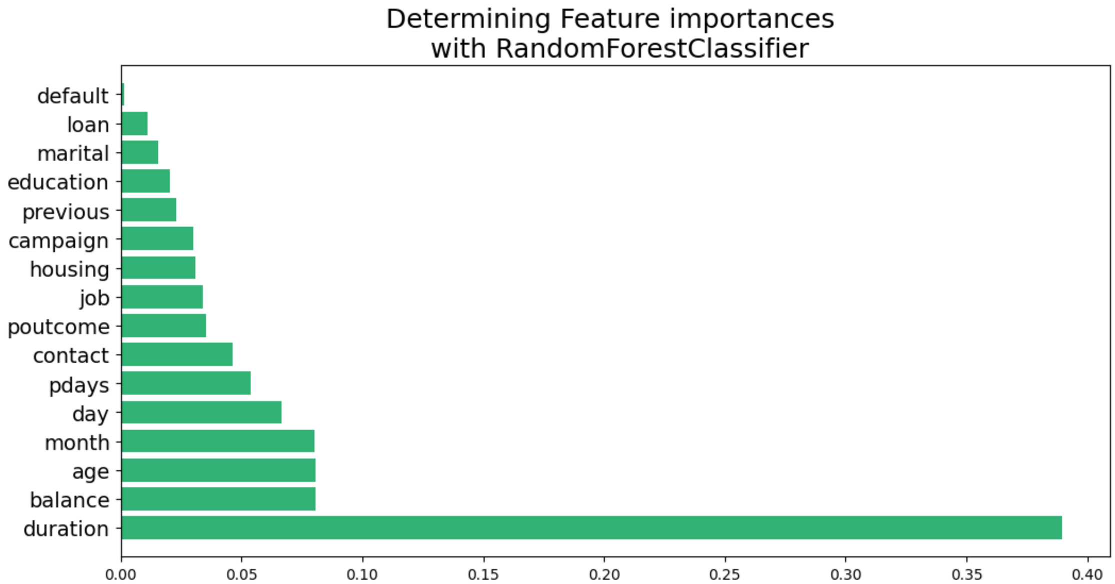
0.0.6 4c Continued for Feature importance:

[172]:
```
# Convert the columns into categorical variables
df['job'] = df['job'].astype('category').cat.codes
df['marital'] = df['marital'].astype('category').cat.codes
df['education'] = df['education'].astype('category').cat.codes
df['contact'] = df['contact'].astype('category').cat.codes
df['poutcome'] = df['poutcome'].astype('category').cat.codes
df['month'] = df['month'].astype('category').cat.codes
df['default'] = df['default'].astype('category').cat.codes
df['loan'] = df['loan'].astype('category').cat.codes
df['housing'] = df['housing'].astype('category').cat.codes
df['deposit'] = df['deposit'].astype('category').cat.codes

x = df.drop(columns=["deposit"])
y = df["deposit"]
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.2,stratify=y)
rf_clf = RandomForestClassifier(max_depth= None, min_samples_leaf= 1,␣
 ↪min_samples_split= 5, n_estimators= 100)
rf_clf.fit(x_train,y_train)
y_pred = rf_clf.predict(x_test)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
# Print the results
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print(classification_report(y_test, y_pred))

importances = rf_clf.feature_importances_
feature_names = x_test.columns
```

```python
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(x_train.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest
def feature_importance_graph(indices, importances, feature_names):
    plt.figure(figsize=(12,6))
    plt.title("Determining Feature importances \n with RandomForestClassifier",␣
 ↪fontsize=18)
    plt.barh(range(len(indices)), importances[indices], color='#31B173', ␣
 ↪align="center")
    plt.yticks(range(len(indices)), feature_names[indices],␣
 ↪rotation='horizontal',fontsize=14)
    plt.ylim([-1, len(indices)])

feature_importance_graph(indices, importances, feature_names)
plt.show()
```

```
Precision: 0.8254385964912281
Recall: 0.889413988657845
F1 Score: 0.8562329390354868
              precision    recall  f1-score   support

           0       0.89      0.83      0.86      1175
           1       0.83      0.89      0.86      1058

    accuracy                           0.86      2233
   macro avg       0.86      0.86      0.86      2233
weighted avg       0.86      0.86      0.86      2233

Feature ranking:
1. feature 11 (0.389672)
2. feature 5 (0.080905)
3. feature 0 (0.080606)
4. feature 10 (0.080358)
5. feature 9 (0.066613)
6. feature 13 (0.053780)
7. feature 8 (0.046544)
8. feature 15 (0.035253)
9. feature 1 (0.033910)
10. feature 6 (0.030800)
11. feature 12 (0.029987)
12. feature 14 (0.022965)
```

```
13. feature 3 (0.020260)
14. feature 2 (0.015718)
15. feature 7 (0.011101)
16. feature 4 (0.001526)
```



From the above graph. once can see that, duration is the most important feature that defines whether a customer decides to open a term deposit or not.