Bank Term Deposit Prediction

A Machine Learning approach

Jeevasurya V 1934013

Kishore Kumar R 1934018

1 Aim:

The following project focus on the analysis of a dataset 'Bank Marketing' which contains data about customers and aims to get useful insights from it and predict if a customer will accept a deposit offer or not.

2 Dataset Used:

Bank marketing dataset uploaded originally in the UCI Machine Learning Repository. The dataset gives information about a marketing campaign of a financial institution.

Numeric - age, balance, campaign, pdays, previous
Categorical - marital, education, default (credit), housing, loan, contact, month, day_of_week, duration, postcome
Target - deposit.

3 Dataset Cleaning:

The data is read from Google Drive and different preprocessing were done.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
%matplotlib inline
```

[2]:
```
warnings.simplefilter("ignore")
```

[3]:
```
data = pd.read_csv("/content/drive/MyDrive/Econometrics/bank.csv")
```

[4]:
```
df = data.copy()
```

[5]:
```
data.head()
```

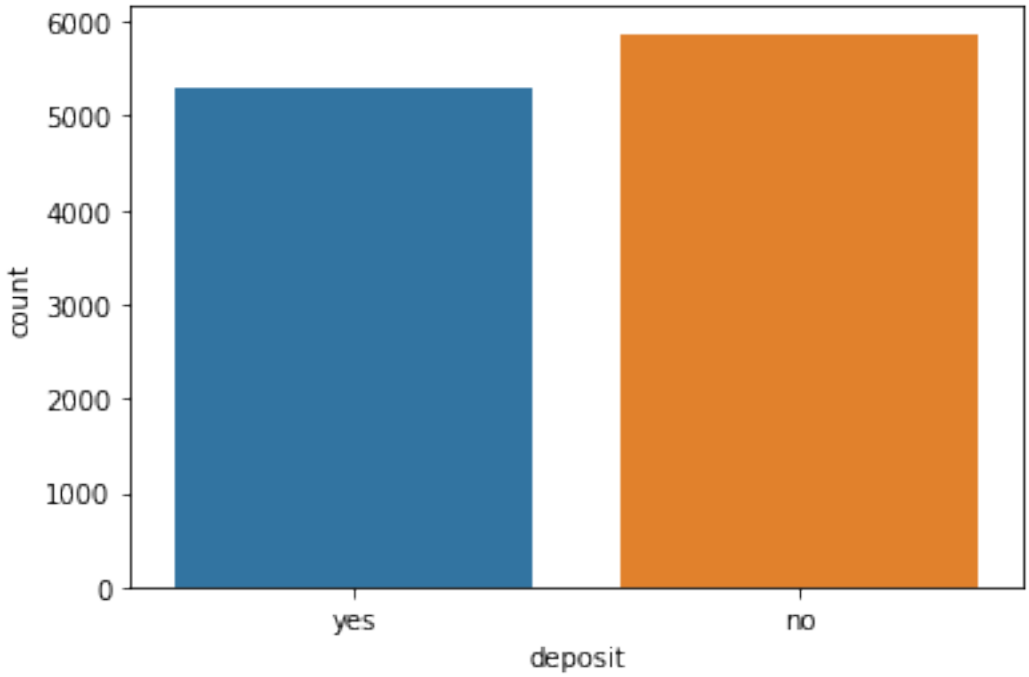
[5]:
```
   age          job  marital  education  ... pdays  previous poutcome deposit
0   59       admin.  married  secondary  ...    -1         0  unknown     yes
1   56       admin.  married  secondary  ...    -1         0  unknown     yes
2   41   technician  married  secondary  ...    -1         0  unknown     yes
3   55     services  married  secondary  ...    -1         0  unknown     yes
4   54       admin.  married   tertiary  ...    -1         0  unknown     yes

[5 rows x 17 columns]
```

[6]:
```
df.shape
```

[6]:
```
(11162, 17)
```

[7]:
```
sns.countplot(x = "deposit",data = data)
```

[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb758771d10>
```



The dataset is well balanced.

[8]:
```
data.describe()
```

[8]:
```
                age       balance  ...          pdays      previous
count  11162.000000  11162.000000  ...  11162.000000  11162.000000
mean      41.231948   1528.538524  ...     51.330407      0.832557
std       11.913369   3225.413326  ...    108.758282      2.292007
min       18.000000  -6847.000000  ...     -1.000000      0.000000
25%       32.000000    122.000000  ...     -1.000000      0.000000
50%       39.000000    550.000000  ...     -1.000000      0.000000
75%       49.000000   1708.000000  ...     20.750000      1.000000
```

```
max       95.000000  81204.000000  ...    854.000000     58.000000

[8 rows x 7 columns]
```

[9]:
```
data.info()
```

```
<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
```

[10]:
```
data["day"] = data["day"].astype('object')
```

Day is given as int. Day must be an object type.

[11]:
```
data.isnull().sum()
```

[11]:
```
age          0
job          0
marital      0
education    0
default      0
balance      0
housing      0
loan         0
contact      0
day          0
month        0
duration     0
```

```
campaign     0
pdays        0
previous     0
poutcome     0
deposit      0
dtype: int64
```

[12]:
```
data[data.duplicated()]
```

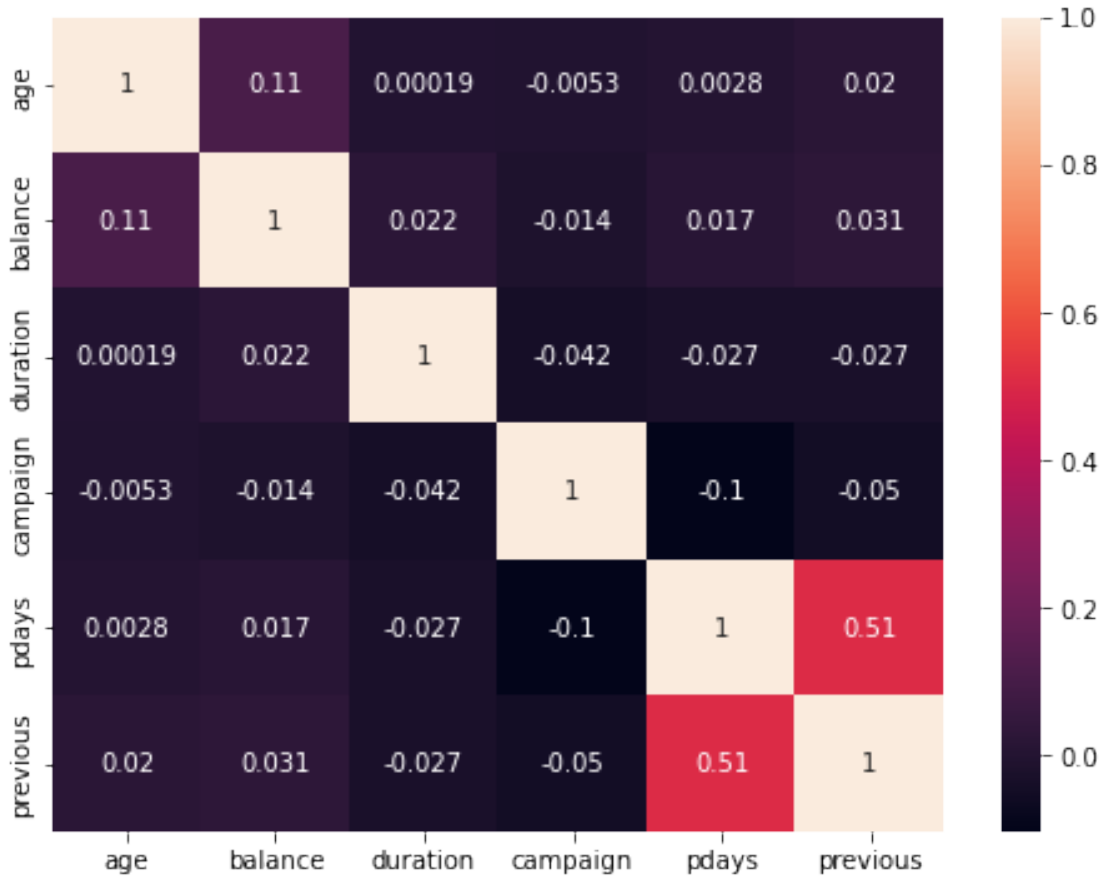
[12]:
```
Empty DataFrame
Columns: [age, job, marital, education, default, balance, housing, loan,
contact, day, month, duration, campaign, pdays, previous, poutcome, deposit]
Index: []
```

There are no null values in the dataset. Also there are no duplicate values present in it. We can go down to Exploratory Data Analysis.

4 Exploratory Data Analysis

[13]:
```
plt.figure(figsize = (8,6))
sns.heatmap(data.corr(),annot = True)
```

[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb746ad3950>
```

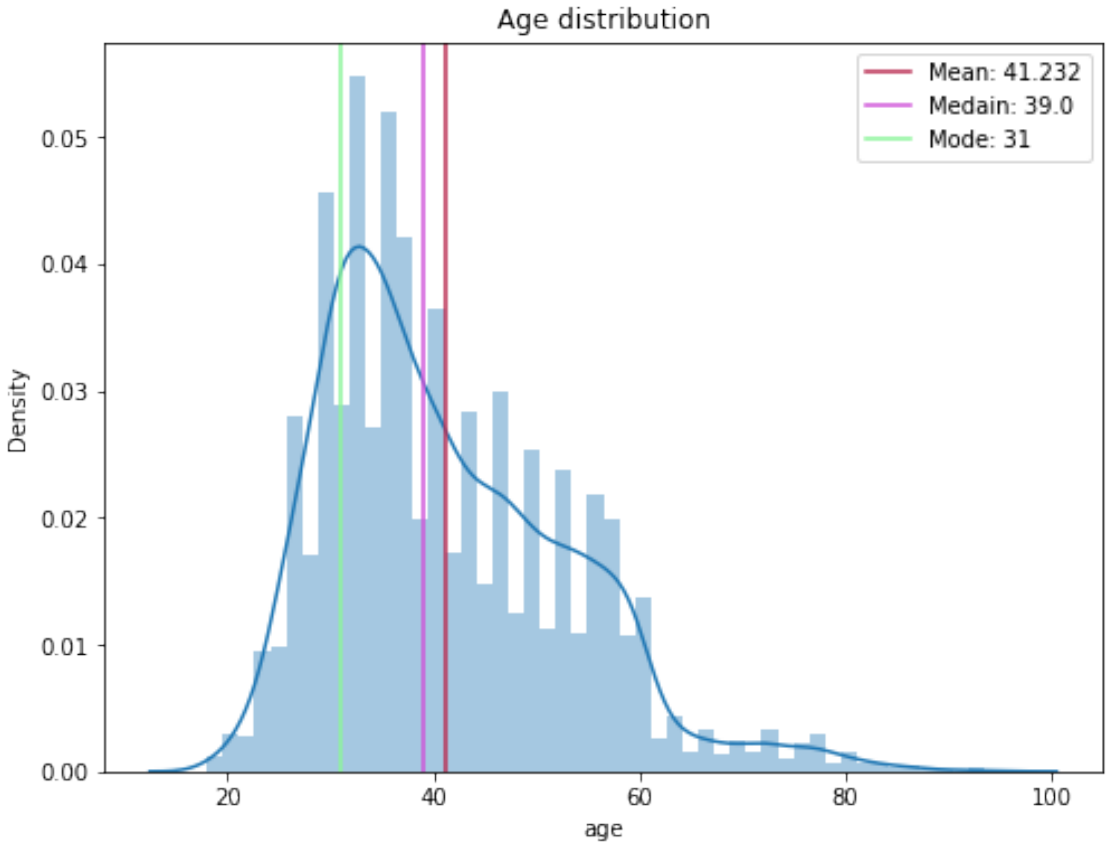


- There is no significant inter correlation among the variables, other than pdays and previous.
- That is expected, as it is mentioned that pdays = -1 means that the customer have not been contaced yet and previous will be 0.

[14]:
```
def analyzeContinuous(col,title,df = data):
  plt.figure(figsize = (8,6))
  mean = np.round(df[col].mean(),3)
  median = np.round(df[col].median(),3)
  mode = df[col].mode()[0]

  sns.distplot(df[col])
  plt.axvline(mean,color = '#ba2d52',label = "Mean: " + str(mean))
  plt.axvline(median,color = '#d052d9',label = "Medain: " + str(median))
  plt.axvline(mode,color = '#8cf59a',label = "Mode: " + str(mode))
  plt.title(title)
  plt.legend()
```

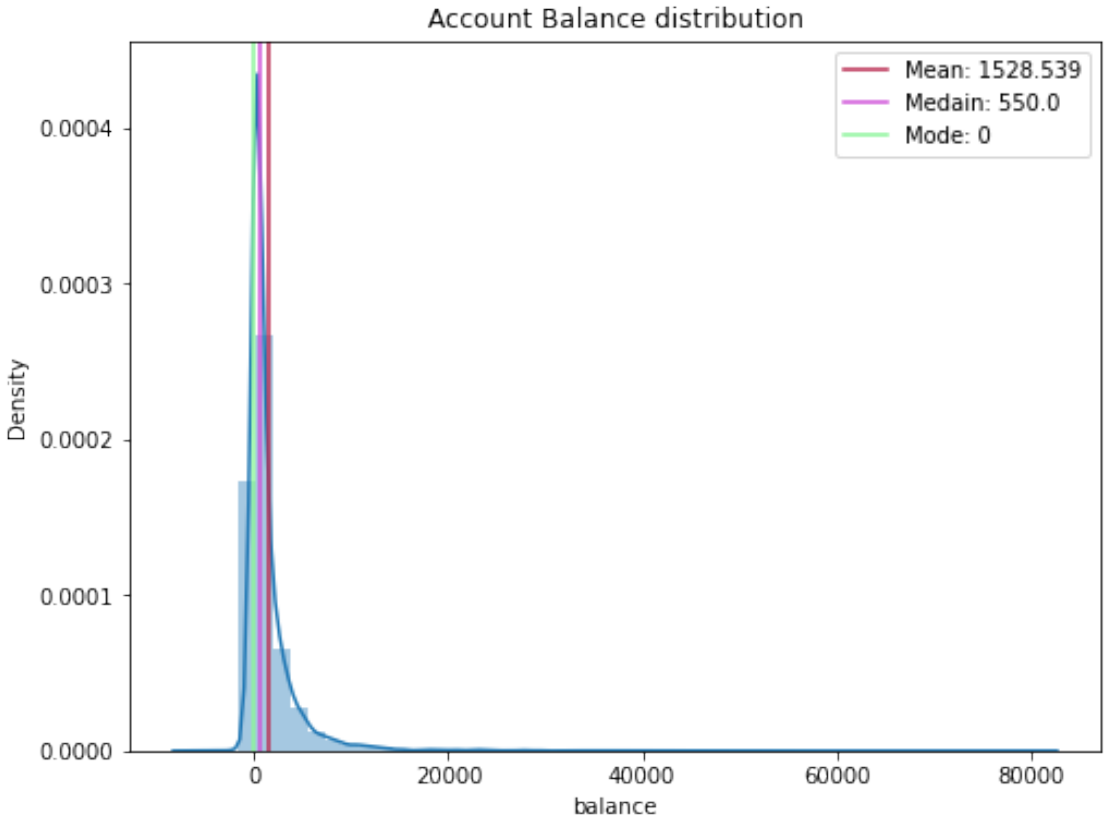
The Function above is used for creating plots for continuous values. The function can be called with any continuous value present in the data, to get its distribution plots.

[15]:
```
analyzeContinuous("age","Age distribution")
```



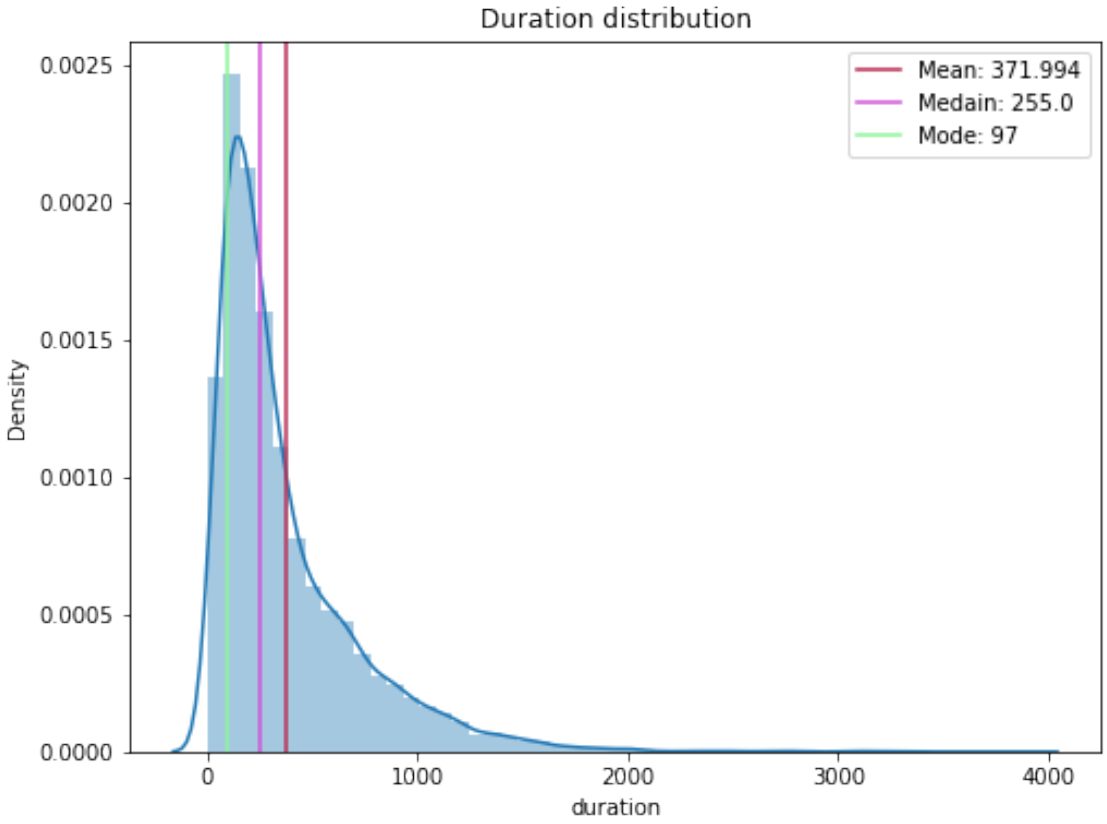
Age is Skewed to the right. The mean and median age are close but mode is low. People the age of 20 and above are contacted. People at the age 32 are contacted more frequently. People above the age of 60 is not contacted as frequent as other young customers.

[16]:
```
analyzeContinuous("balance","Account Balance distribution")
```



Balance is highly right skewed. The mean balance is 1528. There are few people with high balance amount, this is why the right tail goes around 80000 and few people are having negative balance. Most people have 0 balance.

[17]:
```
analyzeContinuous("duration","Duration distribution")
```



Most call durations are 97 seconds. Suggesting most people end the call within the first 2 minutes of the campaign. The average time a customer is engaged is 6 minutes.

[18]:
```
print(f" New Customers contaced : {sum(data['pdays'] == -1)}")
print(f" New Customers percentage : {np.round(sum(data['pdays'] == -1)/data.
 ↪shape[0] * 100,3)} %")
```

```
New Customers contaced : 8324
New Customers percentage : 74.574 %
```

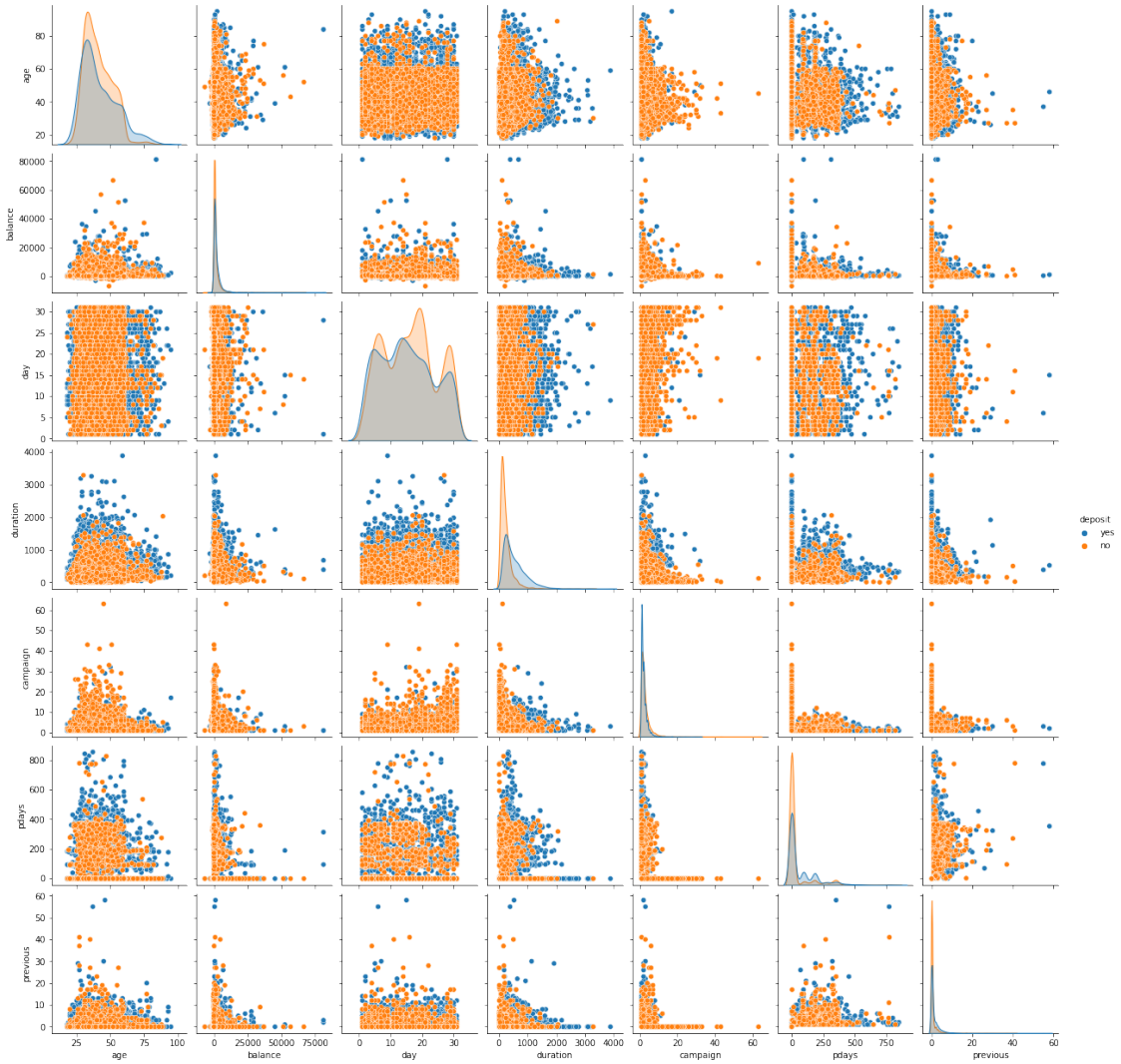
Almost 75 % of the contacted persons are contacted for the first time.

[19]:
```
def analyzeCategorical(col,title = "",df = data):
  plt.figure(figsize = (8,6))
  sns.countplot(x = col,hue = "deposit",data = df)
  plt.xticks(rotation = 45)
  plt.title(title)
```

This function is used for plotting categorical data.

[20]:
```
sns.pairplot(data,hue = "deposit")
```

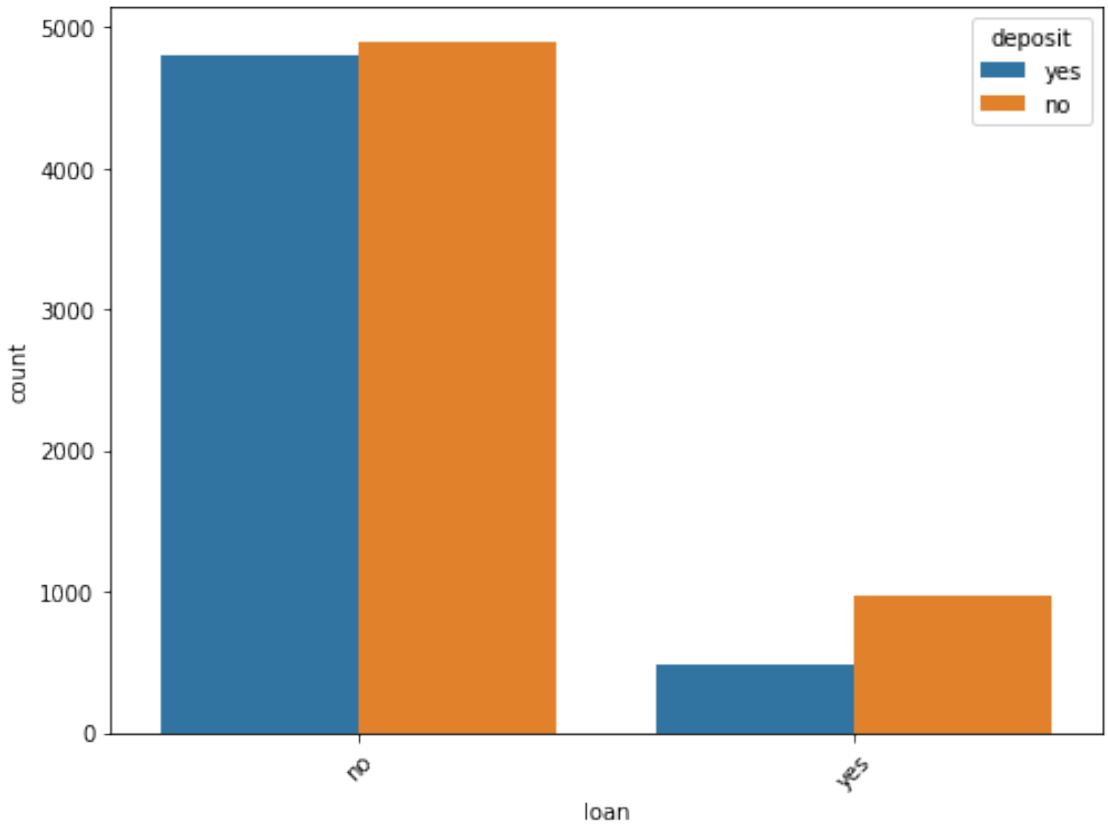
[20]:
```
<seaborn.axisgrid.PairGrid at 0x7fb73b080f90>
```



- Most of the plots in pair plot are overlapping.
- That is both yes and no response have similar distribution among continuous variables.
- Few Yes classes have higher values in some of the pairs.
- It is difficult to draw inferences from this pair plot

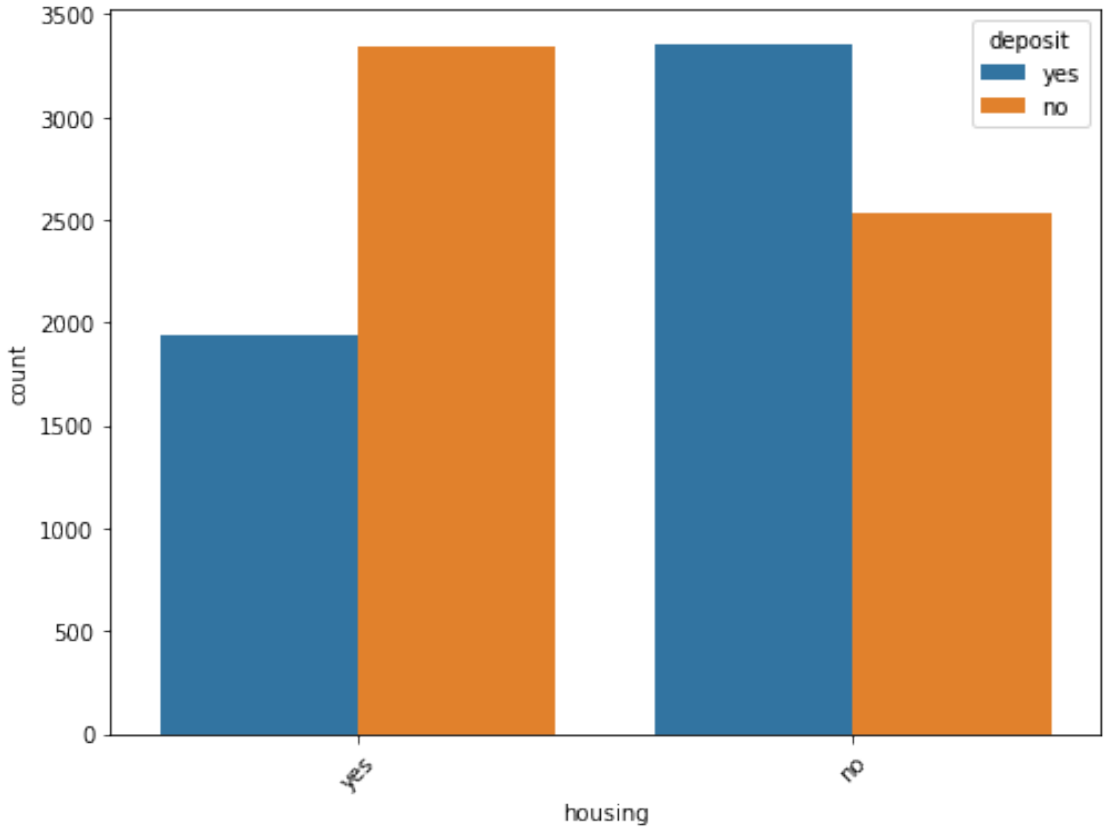
Since loan is one of the key factors, lets see how it affects the deposit.

[21]:
```
analyzeCategorical("loan")
```



Customers with personal loan tends to reject the deposit more.

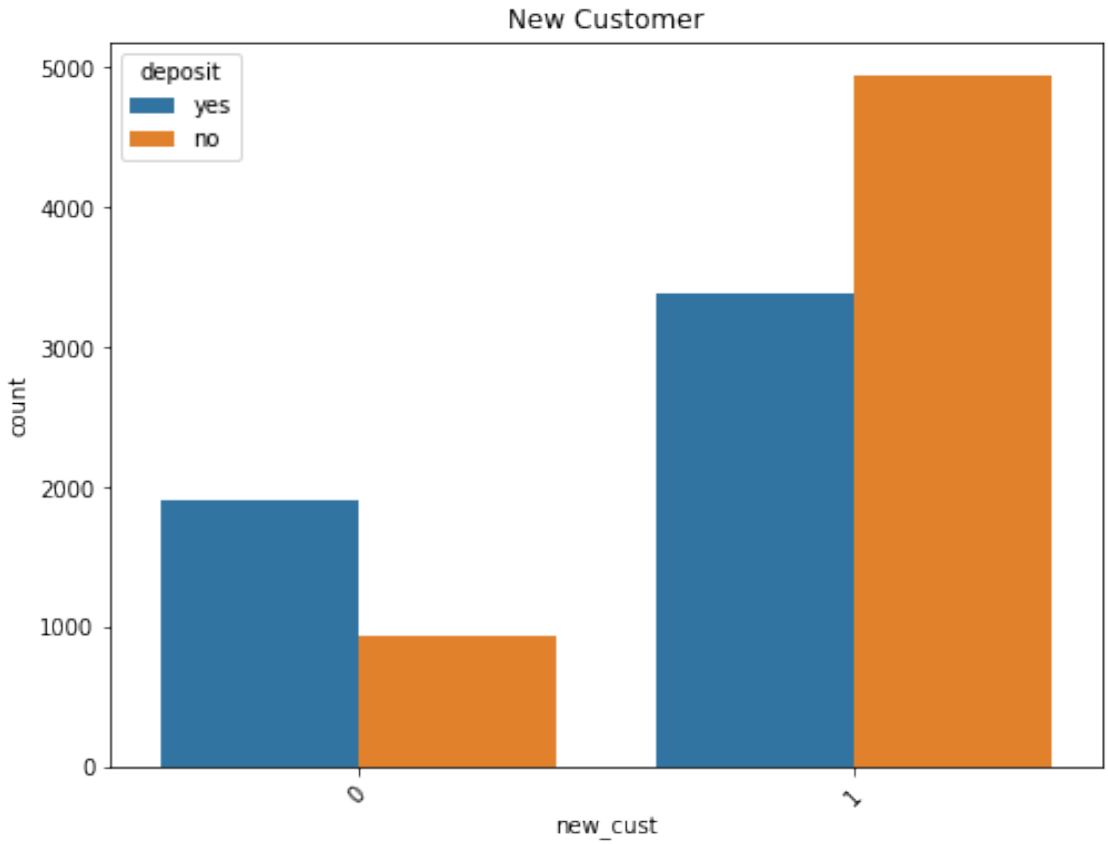
[22]:
```
analyzeCategorical("housing")
```



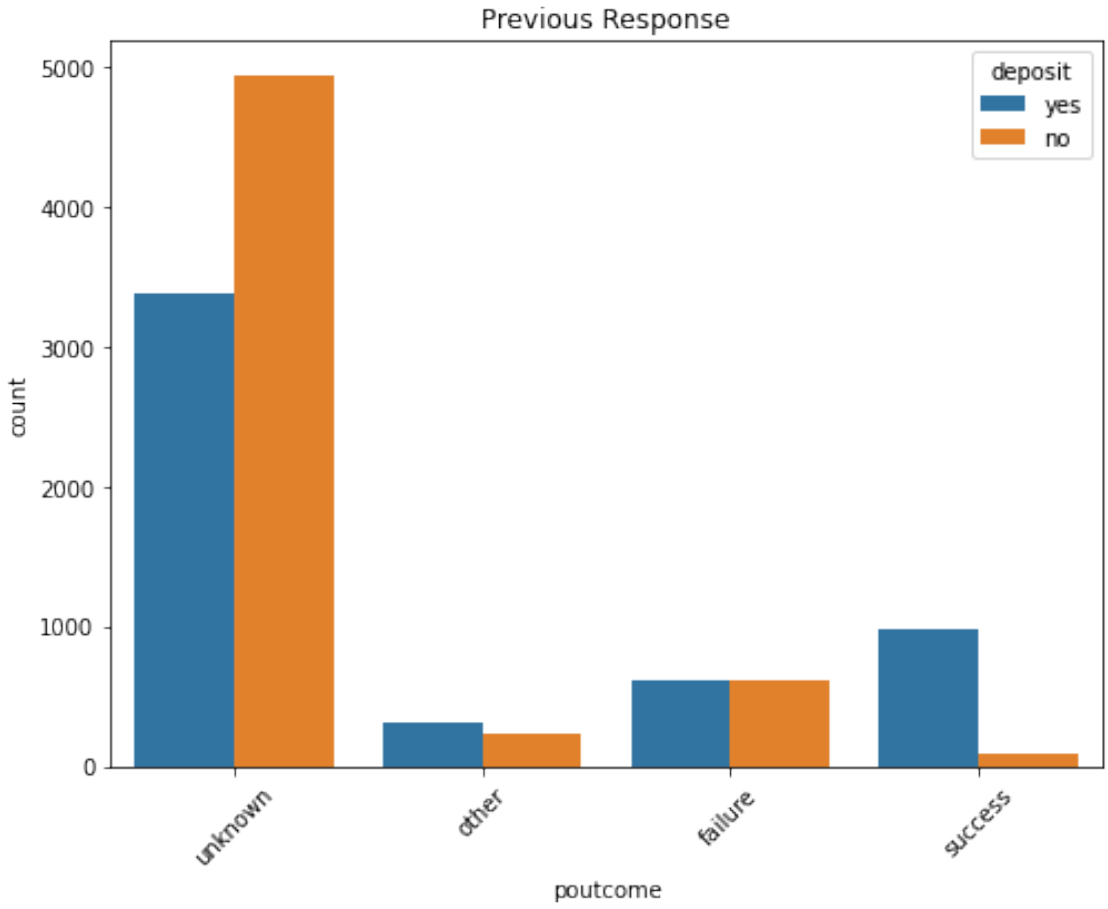
The same is the case with Housing loan. People with housing loan tends to reject the deposit.

[23]:
```
new_cust = data["pdays"] == -1
new_cust = new_cust.map({True:1,False:0})
data["new_cust"] = new_cust
```

[24]:
```
analyzeCategorical("new_cust","New Customer")
```

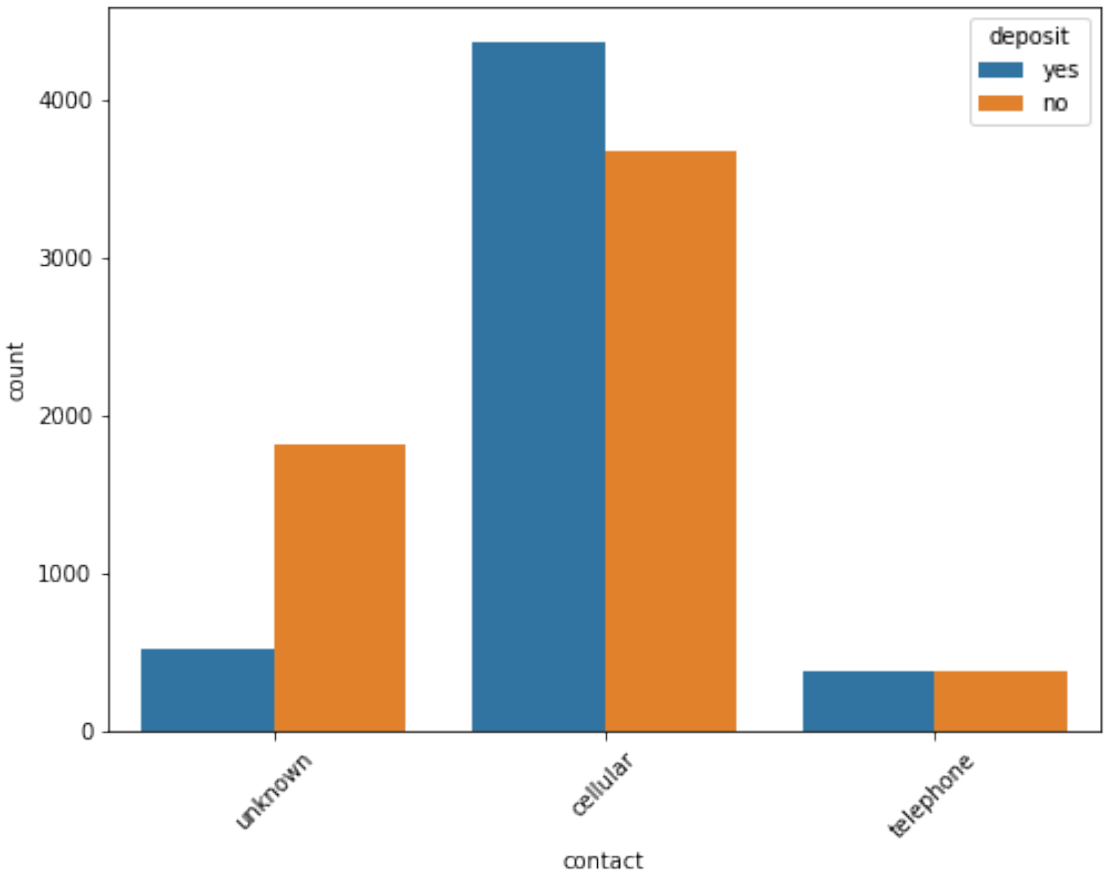


[25]:
```
analyzeCategorical("poutcome","Previous Response")
```



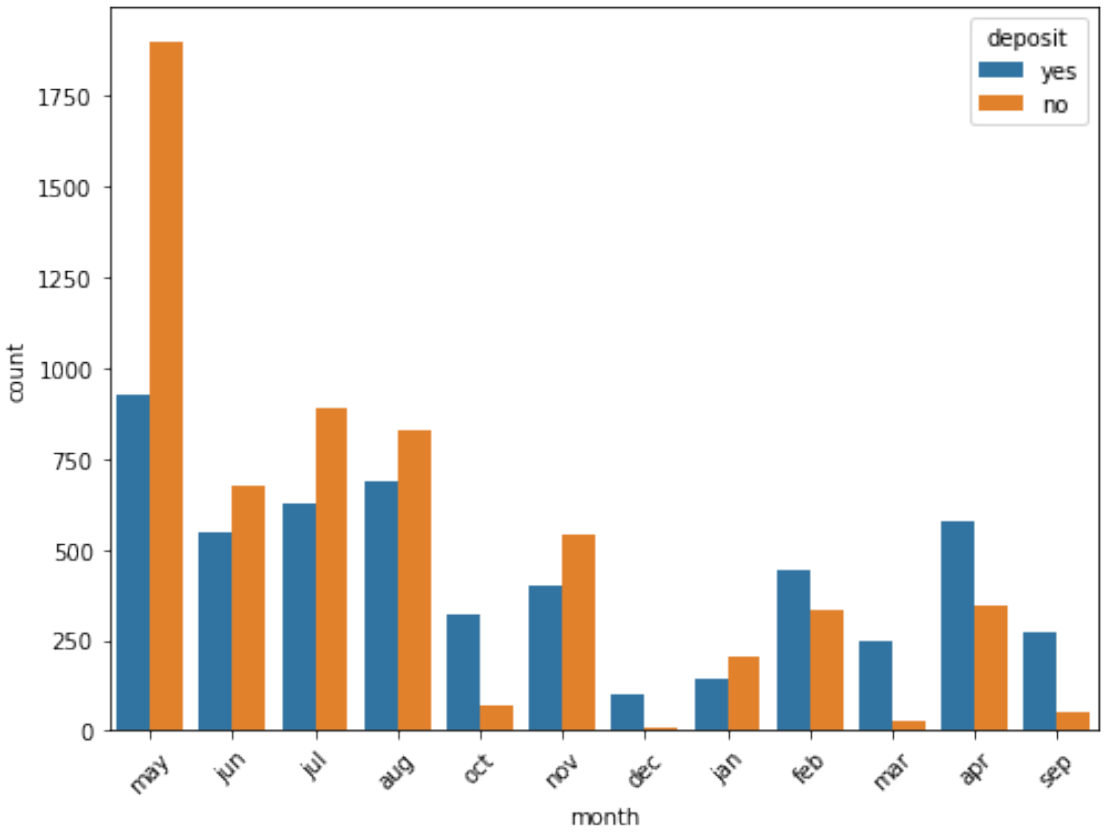
New customers are more rejecting. Customers who where previously contacted seems to again accept the deposit. Also people who deposited during the last contact tends to again deposit for the term.

[26]:
```
analyzeCategorical("contact")
```



Customers who were contacted via cellular are more accepting.

[27]:
```
analyzeCategorical("month")
```

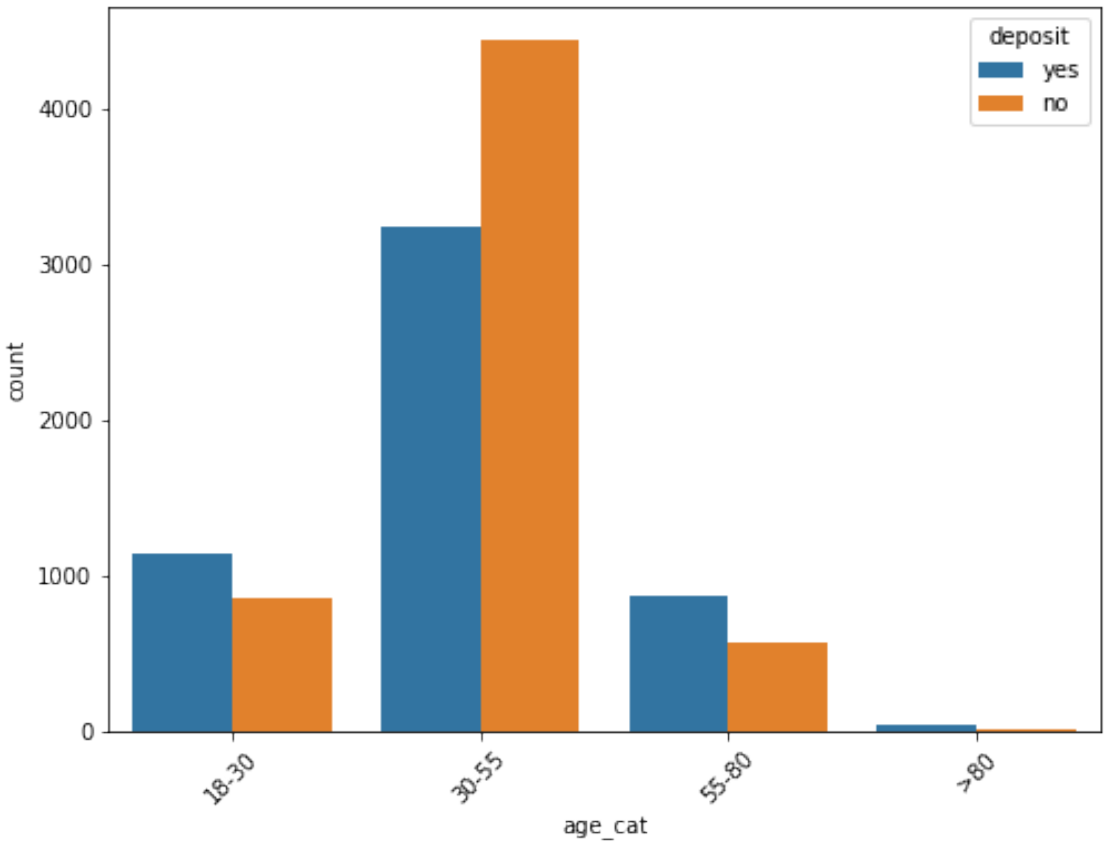


People accept the term deposit more in the months of February, March, April, September, October, and December

Let's see how people within an age group respond

[28]:
```
age_cat = pd.cut(data["age"],bins = [17,30,55,80,np.inf],labels =␣
↪["18-30","30-55","55-80",">80"])
data["age_cat"] = age_cat
```

[29]:
```
analyzeCategorical("age_cat")
```



Most people in their middle age are rejecting deposit. Young people and old(retired) people are more accepting the deposit.

4.1 Preprocessing

[30]:
```
from sklearn.preprocessing import StandardScaler
```

[31]:
```
df['deposit'] = df['deposit'].map({'yes':1,'no':0})

processed_data = pd.get_dummies(df,drop_first = True)

scaler=StandardScaler()
```

- Encoding 'Yes' and 'No' to 1 and 0.
- One hot encoding is done on categorical variables using get_dummies.
- This increases the no of features to 41 (after dropping deposit and pdays)

[32]:
```
processed_data.head()
```

[32]:
```
   age  balance  day  ...  poutcome_other  poutcome_success  poutcome_unknown
0   59     2343    5  ...               0                 0                 1
```

```
1   56       45    5  ...                0                0                1
2   41     1270    5  ...                0                0                1
3   55     2476    5  ...                0                0                1
4   54      184    5  ...                0                0                1

[5 rows x 43 columns]
```

This is the final processed data.

[33]:
```
np.random.seed(142)
X = processed_data.drop(['deposit','pdays'],axis = 1)
y = processed_data['deposit']
X = scaler.fit_transform(X)
```

The processed data is scaled to have similar range of values. Scaling will help in improving the accuracy of the model.

[34]:
```
#spliting the dataset
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.
 ↪33,random_state=4)
```

The data is then split into test and train. The test size is 33% of the original data.

5 Model Building

Two models were built on the dataset. First a Decision tree was fit on the data. Then required changes were done accordingly. After that a Logistic regression model was built. The models are also evaluated using metrics.

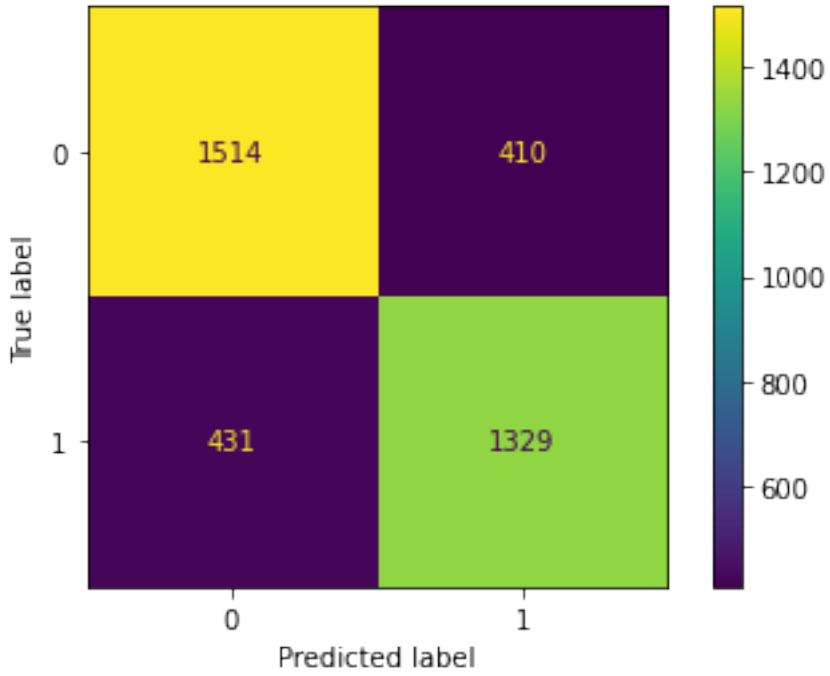
[35]:
```
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
```

5.1 Decision Tree

[36]:
```
dt=DecisionTreeClassifier()

dt.fit(X_train,y_train)
print("Train Score: ",dt.score(X_train,y_train)*100)
print("Test Score: ",dt.score(X_test,y_test)*100)

y_pred = dt.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
```

```
ConfusionMatrixDisplay(cm).plot()
print("Report: \n", classification_report(y_test, y_pred))
print("Cross Validation Score : ", cross_val_score(dt, X_test, y_test, cv=10).
 ↪mean())
```

```
Train Score:  100.0
Test Score:  77.171552660152
Report:
               precision    recall  f1-score   support

           0       0.78      0.79      0.78      1924
           1       0.76      0.76      0.76      1760

    accuracy                           0.77      3684
   macro avg       0.77      0.77      0.77      3684
weighted avg       0.77      0.77      0.77      3684

Cross Validation Score :  0.774703222575704
```



Inference

- The model is overfitting.
- The train score is 100 % and test score is 78.6 %.
- The model has learned the patterns in train data and tends to fit it well.
- Thus it fails to predict propelry in unseen, new data.

We can use HyperParameter Tuning algorithms, like GridSearchCV and RandomSearchCV for finding the best parameters that will give good accuracy for the model.

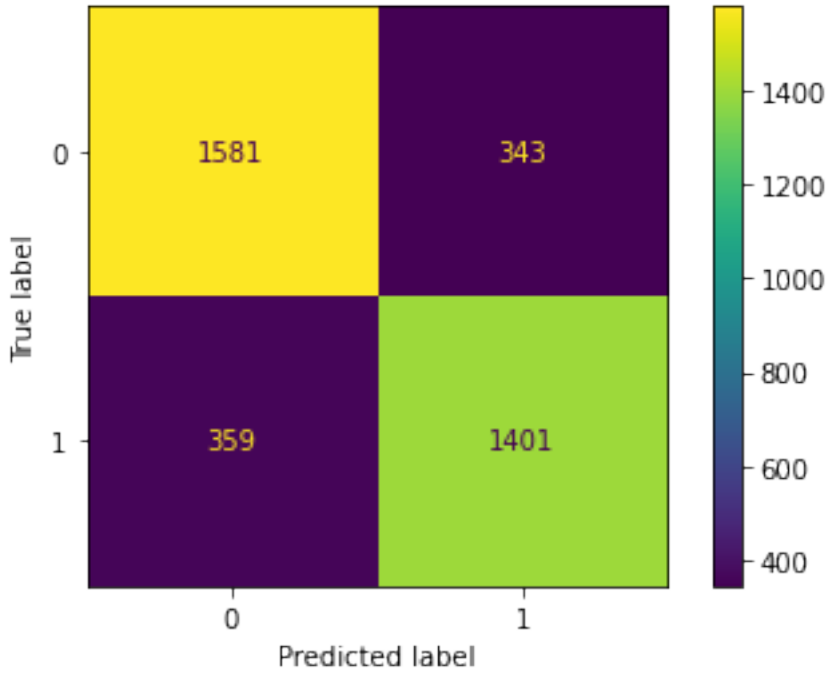
[37]:
```
from sklearn.model_selection import GridSearchCV

# We pass this various parameters for the GridSearchCV algorithm to find the␣
 ↪best fitting parameters
param_grid = {
    "max_depth": [3,5,10,15,20,None],
    "min_samples_split": [2,5,8,10,None],
    "min_samples_leaf": [1,2,5,None],
    "max_features": [2,4,6,None]
}

clf = DecisionTreeClassifier()
grid_cv = GridSearchCV(clf, param_grid, scoring="accuracy", n_jobs = -1, cv =␣
 ↪5).fit(X_train, y_train)

print("Best Parameters from the GridSearch", grid_cv.best_params_)
print("Train score for GridSearch: ", accuracy_score(y_train, grid_cv.
 ↪predict(X_train)))
print("Test score for GridSearch: ", accuracy_score(y_test, grid_cv.
 ↪predict(X_test)))
print("Cross Validation score: ", grid_cv.best_score_)
```

```
Best Parameters from the GridSearch {'max_depth': 10, 'max_features': None,
'min_samples_leaf': 5, 'min_samples_split': 8}
Train score for GridSearch:  0.8673442096817331
Test score for GridSearch:  0.8094462540716613
Cross Validation score:  0.8159972636059593
```

[38]:
```
print("Train Score: ",grid_cv.score(X_train,y_train)*100)
print("Test Score: ",grid_cv.score(X_test,y_test)*100)

y_pred = grid_cv.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm).plot()
print("Report: \n", classification_report(y_test, y_pred))
print("Cross Validation Score: ", grid_cv.best_score_ * 100)

dt_roc_auc = roc_auc_score(y_test, y_pred)
fpr1, tpr1, thresholds1 = roc_curve(y_test, grid_cv.predict_proba(X_test)[:,1])
```

```
Train Score:  86.73442096817331
Test Score:  80.94462540716613
Report:
               precision    recall  f1-score   support
```

```
              0       0.81      0.82      0.82      1924
              1       0.80      0.80      0.80      1760

    accuracy                            0.81      3684
   macro avg        0.81      0.81      0.81      3684
weighted avg        0.81      0.81      0.81      3684

Cross Validation Score:  81.59972636059592
```



After fitting the decision tree with correct parameters, the model is giving good results. The Difference between test and train score is < 5% and The cross validation scores are almost same.

5.2 Logistic Regression

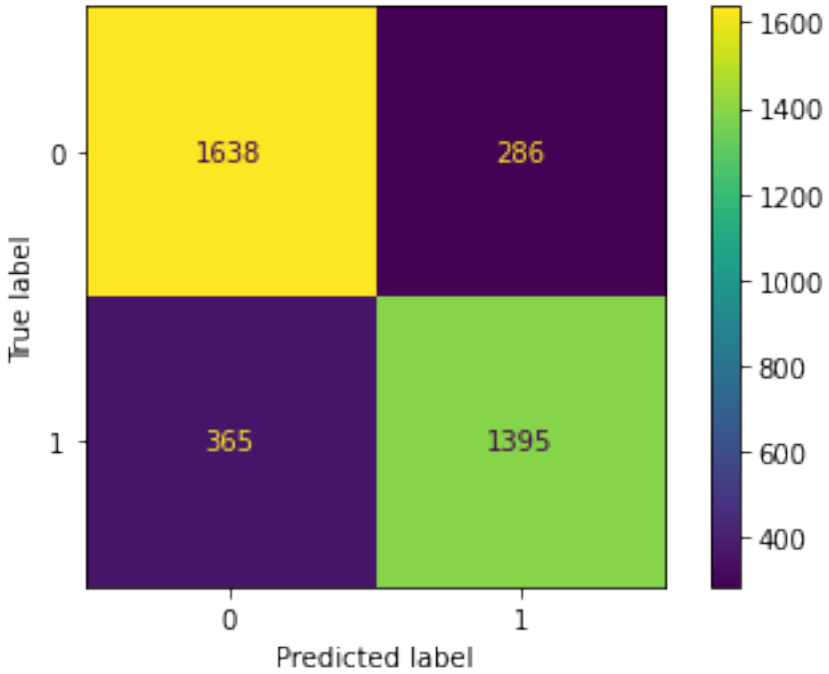
[39]:
```
lr = LogisticRegression()
lr.fit(X_train,y_train)
```

[39]:
```
LogisticRegression()
```

[40]:
```
y_pred = lr.predict(X_test)

print("Train Score: ",lr.score(X_train,y_train)*100)
print("Test Score: ",lr.score(X_test,y_test)*100)

cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm).plot()
```

```
print("Report: \n", classification_report(y_test, y_pred))
print("Cross Validation Score: ", cross_val_score(lr, X_test, y_test, cv = 10).
 ↪mean() * 100)
```

```
Train Score:  82.65579031826692
Test Score:  82.32899022801303
Report:
              precision    recall  f1-score   support

           0       0.82      0.85      0.83      1924
           1       0.83      0.79      0.81      1760

    accuracy                           0.82      3684
   macro avg       0.82      0.82      0.82      3684
weighted avg       0.82      0.82      0.82      3684

Cross Validation Score:  82.27450807116767
```



Logistic regression has similar training and testing score.

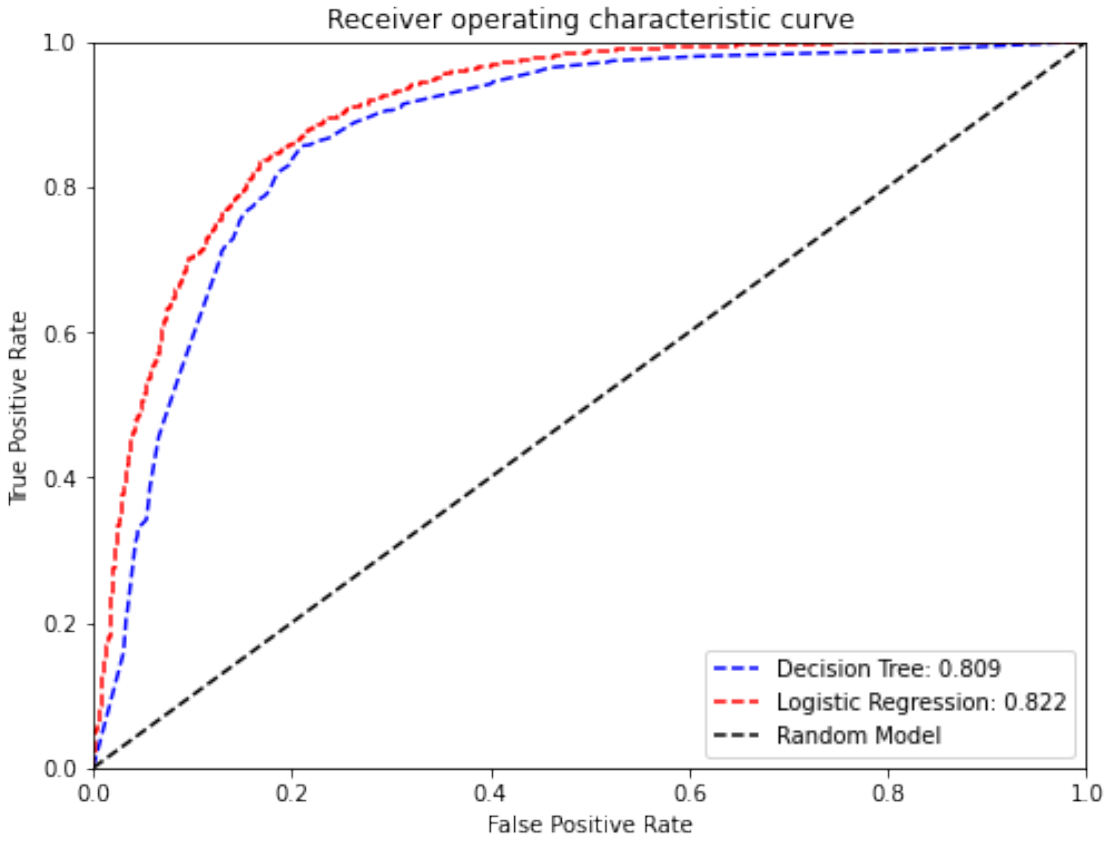
[41]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {
    "max_iter":[100,150,200,250],
    "solver":['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
}
```

```
clf = LogisticRegression()
grid_cv1 = GridSearchCV(clf, param_grid, scoring="accuracy", n_jobs = -1, cv =␣
 ↪5).fit(X_train, y_train)

print("Best Parameters from the GridSearch", grid_cv1.best_params_)
print("Train score for GridSearch: ", accuracy_score(y_train, grid_cv1.
 ↪predict(X_train)))
print("Test score for GridSearch: ", accuracy_score(y_test, grid_cv1.
 ↪predict(X_test)))
print("Cross Validation score: ", grid_cv1.best_score_)
```

```
Best Parameters from the GridSearch {'max_iter': 100, 'solver': 'newton-cg'}
Train score for GridSearch:  0.8265579031826692
Test score for GridSearch:  0.8232899022801303
Cross Validation score:  0.8213451254627726
```

After fitting the models we are comparing the models using ROC curves.

[42]:
```
logit_roc_auc = roc_auc_score(y_test, y_pred)
fpr2, tpr2, thresholds2 = roc_curve(y_test, lr.predict_proba(X_test)[:,1])

plt.figure(figsize = (8,6))
plt.plot(fpr1, tpr1, 'b--', label = f'Decision Tree: {np.round(dt_roc_auc,3)}')
plt.plot(fpr2, tpr2, 'r--', label = f'Logistic Regression: {np.
 ↪round(logit_roc_auc,3)}')
plt.plot([0, 1], [0, 1],'k--',label = 'Random Model')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.title("Receiver operating characteristic curve")
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.show()
```



- The Models perofrm well, Logistic Regression performs better than Decision Tree.
- The AUC tells the ability of the algorithms to distinguish between the two classes.
- Both model have good AUC and can well distinguish between the classes.

5.3 Extracting Decision Rules

[43]:
```
dtree = DecisionTreeClassifier(max_depth = 10, max_features = None,␣
 ↪min_samples_leaf =  5, min_samples_split = 2)
dtree.fit(X_train,y_train)
```

[43]:
```
DecisionTreeClassifier(max_depth=10, min_samples_leaf=5)
```

[44]:
```
# https://mljar.com/blog/extract-rules-decision-tree/
def get_rules(tree, feature_names, class_names):
    tree_ = tree.tree_
    feature_name = [
        feature_names[i] if i != _tree.TREE_UNDEFINED else "undefined!"
        for i in tree_.feature
    ]

    paths = []
```

```
    path = []

    def recurse(node, path, paths):

        if tree_.feature[node] != _tree.TREE_UNDEFINED:
            name = feature_name[node]
            threshold = tree_.threshold[node]
            p1, p2 = list(path), list(path)
            p1 += [f"({name} <= {np.round(threshold, 3)})"]
            recurse(tree_.children_left[node], p1, paths)
            p2 += [f"({name} > {np.round(threshold, 3)})"]
            recurse(tree_.children_right[node], p2, paths)
        else:
            path += [(tree_.value[node], tree_.n_node_samples[node])]
            paths += [path]

    recurse(0, path, paths)

    # sort by samples count
    samples_count = [p[-1][1] for p in paths]
    ii = list(np.argsort(samples_count))
    paths = [paths[i] for i in reversed(ii)]

    rules = []
    for path in paths:
        rule = "if "

        for p in path[:-1]:
            if rule != "if ":
                rule += " and "
            rule += str(p)
        rule += " then "
        if class_names is None:
            rule += "response: "+str(np.round(path[-1][0][0][0],3))
        else:
            classes = path[-1][0][0]
            l = np.argmax(classes)
            rule += f"class: {class_names[l]} (proba: {np.round(100.
↪0*classes[l]/np.sum(classes),2)}%)"
        rule += f" | based on {path[-1][1]:,} samples"
        rules += [rule]

    return rules
```

The above function is used for getting the Rules of the Fitted Decision Tree.

[45]:
```
feature_names = list(processed_data.columns)
feature_names.remove('pdays')
```

```
feature_names.remove('deposit')
```

[46]:
```
from sklearn.tree import _tree
rules = get_rules(dtree,feature_names,["No","Yes"])
```

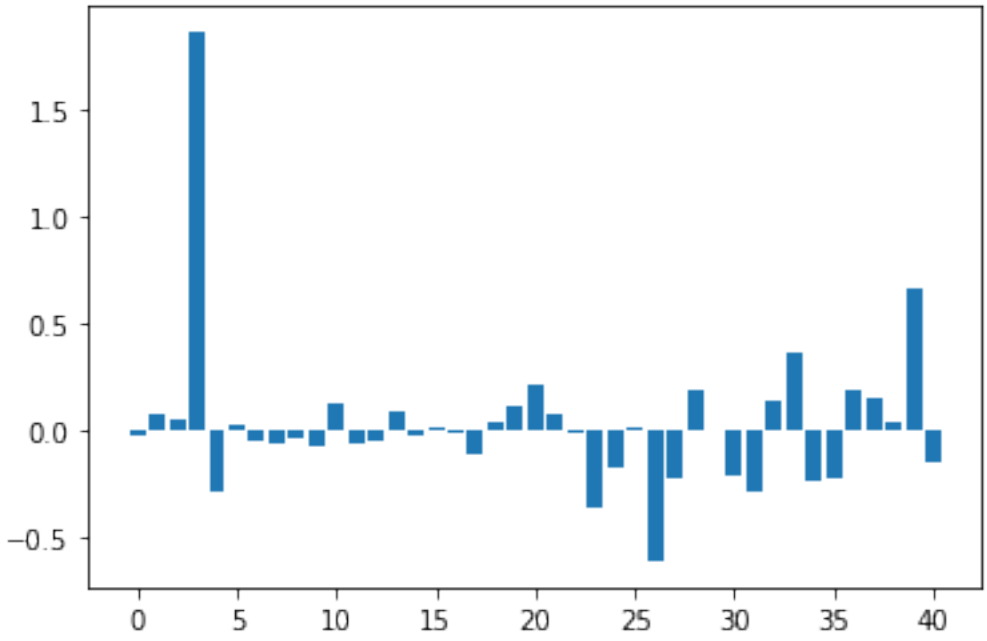
[47]:
```
for i in range(5):
  print(f"Rule {i + 1}: {rules[i]}")
  print()
```

```
Rule 1: if (duration <= -0.477) and (poutcome_success <= 1.372) and (month_mar
<= 3.061) and (month_oct <= 2.525) and (duration <= -0.748) and (month_feb <=
1.693) and (age > -1.488) and (duration <= -0.817) and (month_sep <= 2.829) and
(age <= 1.743) then class: No (proba: 99.04%) | based on 835 samples

Rule 2: if (duration > -0.477) and (duration <= 0.246) and (contact_unknown <=
0.711) and (poutcome_success <= 1.372) and (housing_yes <= 0.054) and (balance >
-0.453) and (loan_yes <= 1.095) and (month_aug <= 1.061) and (month_nov <=
1.494) and (month_jan <= 2.715) then class: Yes (proba: 80.41%) | based on 541
samples

Rule 3: if (duration <= -0.477) and (poutcome_success <= 1.372) and (month_mar
<= 3.061) and (month_oct <= 2.525) and (duration > -0.748) and (housing_yes <=
0.054) and (age > -1.069) and (previous <= 0.291) and (age <= 1.617) and
(balance <= 0.526) then class: No (proba: 84.74%) | based on 439 samples

Rule 4: if (duration <= -0.477) and (poutcome_success <= 1.372) and (month_mar
<= 3.061) and (month_oct <= 2.525) and (duration <= -0.748) and (month_feb <=
1.693) and (age > -1.488) and (duration > -0.817) and (poutcome_other <= 2.112)
and (age > -1.321) then class: No (proba: 94.21%) | based on 328 samples

Rule 5: if (duration > -0.477) and (duration > 0.246) and (duration <= 0.892)
and (contact_unknown <= 0.711) and (poutcome_success <= 1.372) and (housing_yes
<= 0.054) and (loan_yes <= 1.095) and (campaign <= 0.732) and (job_entrepreneur
<= 2.787) and (duration > 0.255) then class: Yes (proba: 82.61%) | based on 299
samples
```

- We are printing out the first 5 rules of the Decision tree.
- Duration is the main feature in all the rules. Followed by Postoutcome.
- These 2 features are mainly contributing for the outcome of the input given.

5.3.1 Feature Importance

[48]:
```
importance = lr.coef_[0]
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```



- From the feature importance plot we can see duration attribute has the highest coefficient.
- Postcome of the last campaign is the second highest.
- Some attributes do have negative coefficients.
- Housing loan makes the most negative impact.

The Feature importance inference and the Decision rules derived, both have similar results.

6 Conclusion:

Both Decision Tree and Logistic regression has similar results. They have similar accuracies and ROC curves. The Decision rules formed were in accord with the feature importance plot and the EDA performed.

To have an successful campaign:

- The target audience must be young age and elderly(retired) people.
- Their previous Outcome to be positive.
- The Customer must be engaged for atleast 6 minutes.