BAN 5753 Mini Case-2

Deposit Opening Classification Problem




Raja Ranjan Bobba **Meghana Singam** **Amani Kolli** **Scott Schwager**

Our goal is to categorize customers based on their likelihood of responding "yes" or "no" to a bank's telephone marketing campaign. The dataset included 41188 observations from May 2008 – November 2010, across 20 different columns. We performed EDA, univariate analysis, bivariate analysis, K-means clustering, and 5 different supervised learning methods on this data to familiarize ourselves with using MLlib within PySpark. From this analysis, we saw that the Gradient-Boosted Tree performed the best of the five supervised learning methods. We also discovered in the bivariate analysis that the duration of a call is an important factor for determining the outcome. Calls that extend for longer time are more likely to result in a deposit than calls that are shorter.

We begin by importing the necessary libraries, initiating a Spark session, and reading in the data. Here, we can see what the raw dataset looks like:

```
In [4]: bankdeposit_df = spark.read \
         .option("header","True")\
         .option("inferSchema","True")\
         .option("sep",",")\
         .csv("D:\\Libraries\\Documents\\BAN 5753\\Week 12\\XYZ_Bank_Deposit_Data_Classification.csv")
        print("There are",bankdeposit_df.count(),"rows",len(bankdeposit_df.columns),
              "columns" ,"in the data.")

        There are 41188 rows 21 columns in the data.
```

```
In [5]: bankdeposit_df.toPandas().head(10)
```

Out[5]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp.var.rat
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...	1	999	0	nonexistent	1.
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...	1	999	0	nonexistent	1.
5	45	services	married	basic.9y	unknown	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.
6	59	admin.	married	professional.course	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.
7	41	blue-collar	married	unknown	unknown	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.
8	24	technician	single	professional.course	no	yes	no	telephone	may	mon	...	1	999	0	nonexistent	1.
9	25	services	single	high.school	no	yes	no	telephone	may	mon	...	1	999	0	nonexistent	1.

10 rows × 21 columns

The first data preprocessing step is to rename all columns that contain a ".", because this character will cause problems later. We will replace it with an underscore:

Rename Columns That Contain "."

In [6]:
```
bankdeposit_df = bankdeposit_df.withColumnRenamed("emp.var.rate","emp_var_rate") \
    .withColumnRenamed("cons.price.idx","cons_price_idx") \
    .withColumnRenamed("cons.conf.idx","cons_conf_idx") \
    .withColumnRenamed("nr.employed","nr_employed")
bankdeposit_df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- month: string (nullable = true)
 |-- day_of_week: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- emp_var_rate: double (nullable = true)
 |-- cons_price_idx: double (nullable = true)
 |-- cons_conf_idx: double (nullable = true)
 |-- euribor3m: double (nullable = true)
 |-- nr_employed: integer (nullable = true)
 |-- y: string (nullable = true)
```

Next, we will encode the Education column into a new integer column. Since levels of education have an implied order, we can provide additional meaning to this column by encoding it into an ordinal variable:

Encode Education To Ordinal Column

In [7]:
```
def udf_multiple(education):
      if (education == 'illiterate'):
        return 1
      elif (education == 'basic.4y'):
        return 2
      elif (education == 'basic.6y'):
        return 3
      elif (education == 'basic.9y'):
        return 4
      elif (education == 'high.school'):
        return 5
      elif (education == 'professional.course'):
        return 6
      elif (education == 'university.degree'):
        return 7
      else: return 0

education_group = udf(udf_multiple)
bankdeposit_df=bankdeposit_df.withColumn("education_group", education_group('education'))
bankdeposit_df=bankdeposit_df.withColumn("education_group",bankdeposit_df.education_group.cast('int'))
```

After doing this, we will look at the various columns, based on their data type. First, we will look at the categorical/string columns:

Identify Categorical & Numerical Columns

In [8]:
```
String_columnList = [item[0] for item in bankdeposit_df.dtypes if item[1].startswith('string')]
Int_columnList = [item[0] for item in bankdeposit_df.dtypes if item[1].startswith('int')]
Double_columnList = [item[0] for item in bankdeposit_df.dtypes if item[1].startswith('double')]

Numerical = Int_columnList + Double_columnList
print("Numerical Columns:",Numerical)
print("String Columns:",String_columnList)
```

```
Numerical Columns: ['age', 'duration', 'campaign', 'pdays', 'previous', 'nr_employed', 'education_group', 'emp_var_rate', 'cons
_price_idx', 'cons_conf_idx', 'euribor3m']
String Columns: ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'day_of_week', 'poutcome',
'y']
```

We also will examine the numerical columns and their statistics:

In [9]:
```
numeric_features = [t[0] for t in bankdeposit_df.dtypes if t[1] in ('int', 'double')]
bankdeposit_df.select(numeric_features).describe().toPandas().transpose()
```

Out[9]:

	0	1	2	3	4
summary	count	mean	stddev	min	max
age	41188	40.02406040594348	10.421249980934045	17	98
duration	41188	258.2850101971448	259.27924883646494	0	4918
campaign	41188	2.567592502670681	2.770013542902324 5	1	56
pdays	41188	962.4754540157328	186.91090734474102	0	999
previous	41188	0.17296299893172767	0.4949010798392901	0	7
emp_var_rate	41188	0.08188550063188148	1.5709597405170317	-3.4	1.4
cons_price_idx	41188	93.57566436831301	0.5788400489541361	92.201	94.767
cons_conf_idx	41188	-40.502600271913245	4.628197856174576	-50.8	-26.9
euribor3m	41188	3.621290812858179	1.7344474048512577	0.634	5.045
nr_employed	41188	5167.0190103913765	72.1780739979935	4964	5228
education_group	41188	4.944037098183937	1.9093116016065543	0	7

After this step, we check for null or missing values in any columns. We can see that there are no null values:

Check For Null Values

In [10]:
```
null_df = bankdeposit_df.select([count(when(col(c).contains('None') | \
                            col(c).contains('NULL') | \
                            (col(c) == '') | \
                            col(c).isNull() | \
                            isnan(c), c
                           )).alias(c)
                    for c in bankdeposit_df.columns])

null_df.toPandas().head(1)
```

Out[10]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	pdays	previous	poutcome	emp_var_rate	cons_price_idx	cons_con
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	

1 rows × 22 columns

We also can check for imbalance between the two classes in the outcome variable, “y”. We see some class imbalance, so we can adjust for this with oversampling if it becomes and issue when we are training and evaluating our models.

Check For Class Imbalance

```
In [11]: bankdeposit_df.groupby('y').count().toPandas()
```

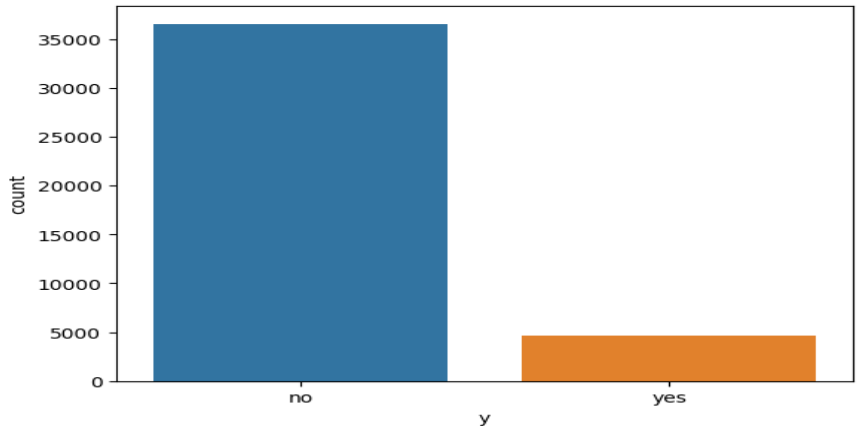
Out[11]:

	y	count
0	no	36548
1	yes	4640

We also can visualize this class imbalance. For many of the visualizations in this report, we will need to convert the PySpark dataframe to Pandas in order to use the Pandas visualization functions, as seen below:

Note: Many Visualizations Require Conversion To Pandas DF

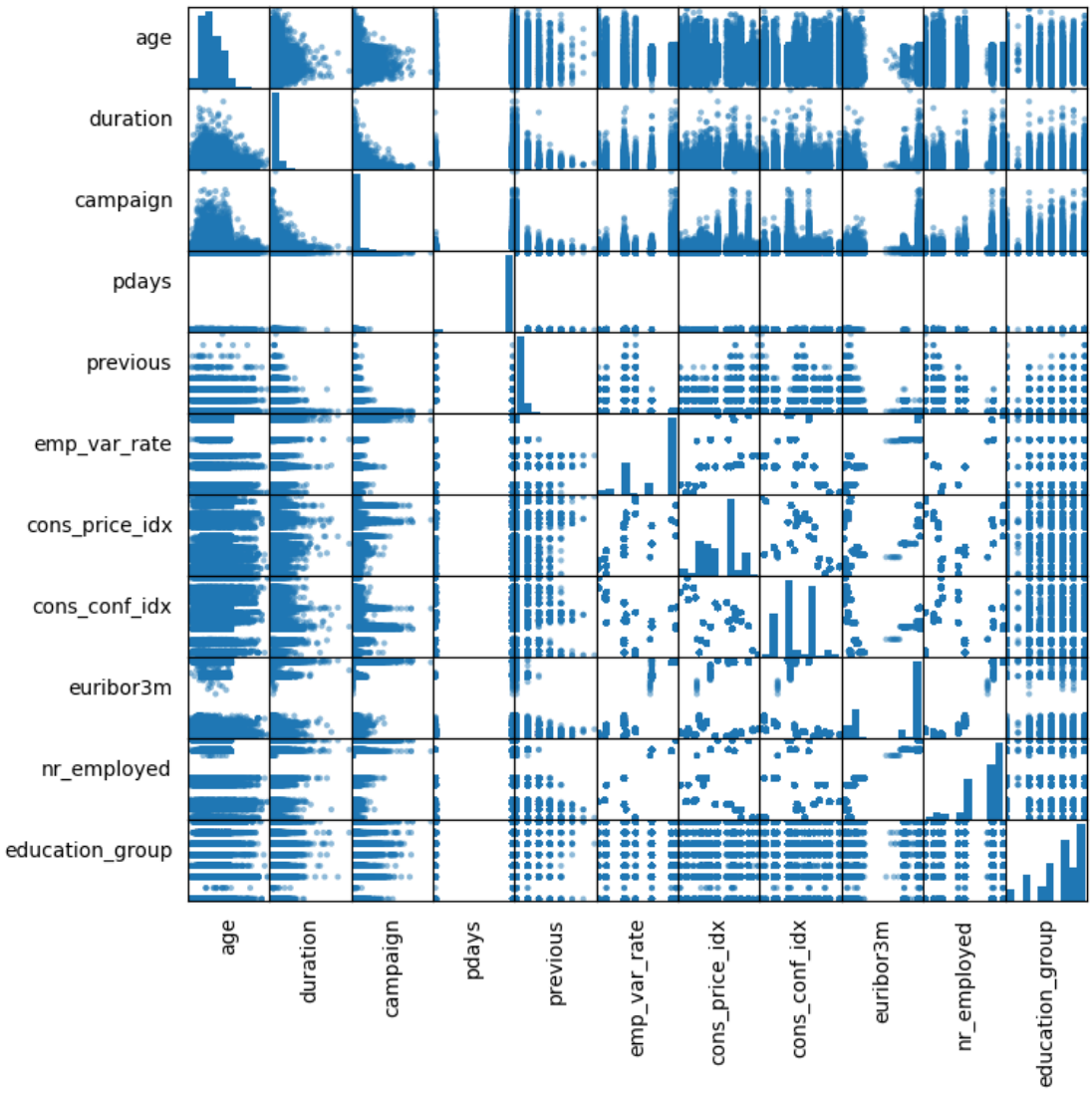
```
In [12]: bank_df = bankdeposit_df.toPandas()
sns.countplot(data=bank_df, x='y')
plt.show()
```



The final step for us to check before we begin more detailed analysis is for correlation between predictor variables. If several predictors are highly correlated, then we can exclude some of them from our analysis without loosing explanation power. Below, our correlation analysis can be seen:

Check For Correlation Between Predictors

```
In [13]: numeric_data = bankdeposit_df.select(numeric_features).toPandas()
axs = pd.plotting.scatter_matrix(numeric_data, figsize=(8, 8));
n = len(numeric_data.columns)
for i in range(n):
    v = axs[i, 0]
    v.yaxis.label.set_rotation(0)
    v.yaxis.label.set_ha('right')
    v.set_yticks(())
    h = axs[n-1, i]
    h.xaxis.label.set_rotation(90)
    h.set_xticks(())
```

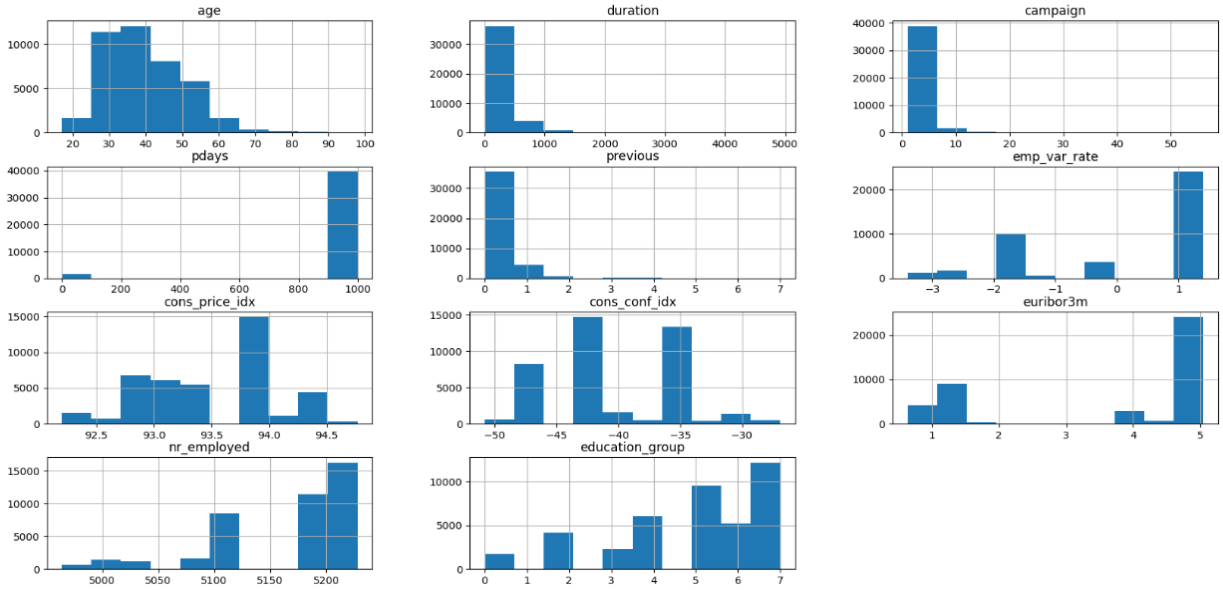


It does not appear that any of our predictor variables are highly correlated with each other, so we can move forward with all the variables included for now.

We now will perform some univariate analysis. First, we will look at the distributions of the numerical variables:

Univariate Analysis

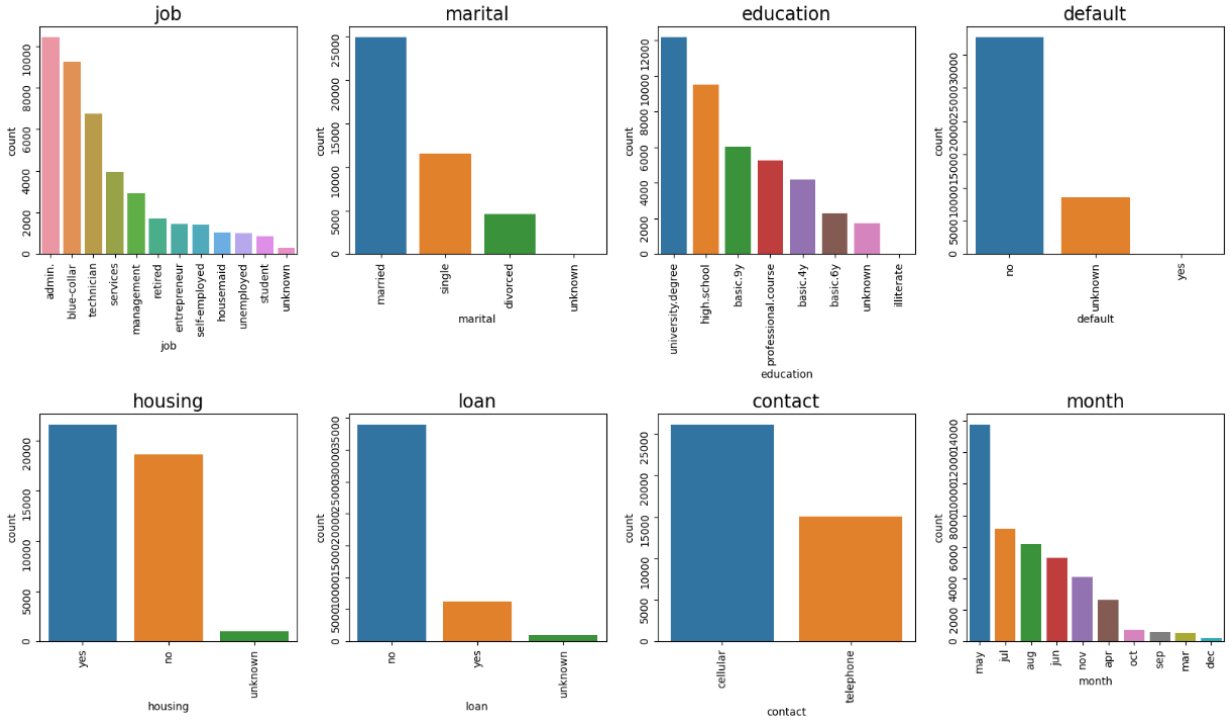
In [14]:
```
numbers = bank_df.select_dtypes(['int64', 'float64']).columns.to_list()
bank_df.hist(figsize=(20,10))
plt.show()

display(bank_df[numbers].describe())
```

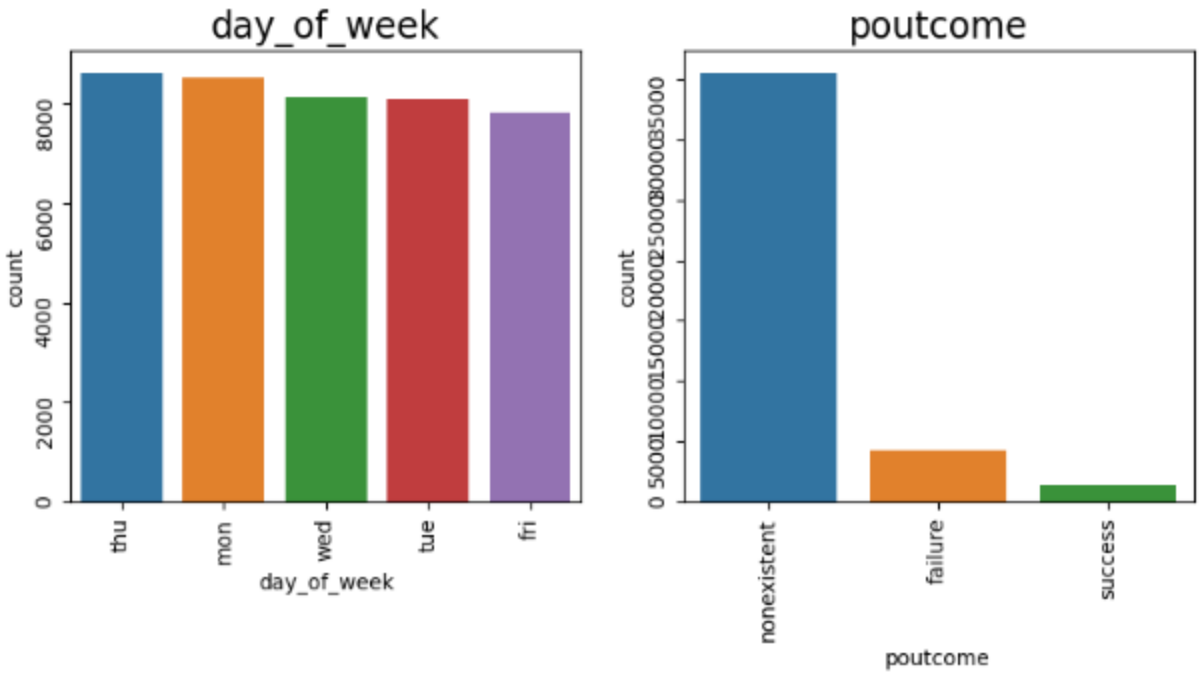


	emp_var_rate	cons_price_idx	cons_conf_idx	euribor3m
count	41188.000000	41188.000000	41188.000000	41188.000000
mean	0.081886	93.575664	-40.502600	3.621291
std	1.570960	0.578840	4.628198	1.734447
min	-3.400000	92.201000	-50.800000	0.634000
25%	-1.800000	93.075000	-42.700000	1.344000
50%	1.100000	93.749000	-41.800000	4.857000
75%	1.400000	93.994000	-36.400000	4.961000
max	1.400000	94.767000	-26.900000	5.045000

We also can look at how the records are distributed for the categorical variables:

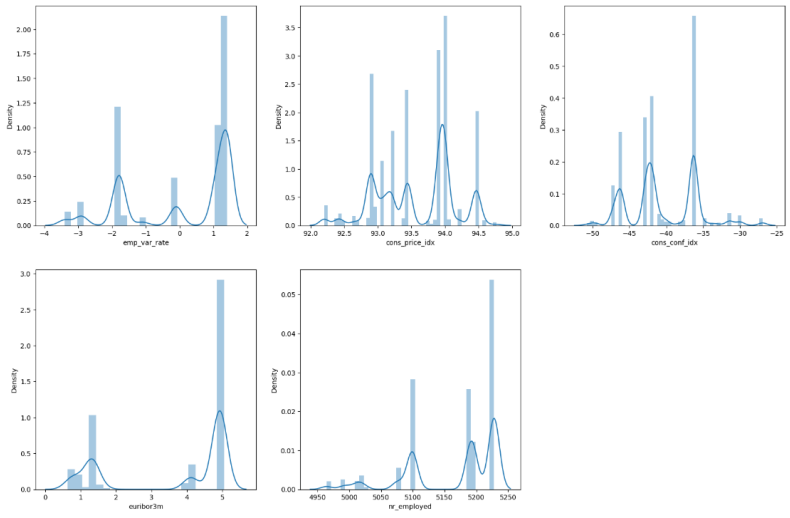
In [15]:
```
fig, ax = plt.subplots(3,4, figsize=(20,17))

cat = bank_df.select_dtypes('object').columns.to_list()
cat = cat[:-1]

ax = ax.ravel()
position = 0

for i in cat:

    order = bank_df[i].value_counts().index
    sns.countplot(data=bank_df, x=i, ax=ax[position], order=order)
    ax[position].tick_params(labelrotation=90)
    ax[position].set_title(i, fontdict={'fontsize':17})

    position += 1

plt.subplots_adjust(hspace=0.7)

plt.show()
```





Finally, for some of the numerical variables, we will attempt to fit a curve to the distribution.

```
plt.subplot(231)
sns.distplot(bank_df['emp_var_rate'])
fig = plt.gcf()
fig.set_size_inches(20,13)

plt.subplot(232)
sns.distplot(bank_df['cons_price_idx'])
fig = plt.gcf()
fig.set_size_inches(20,13)

plt.subplot(233)
sns.distplot(bank_df['cons_conf_idx'])
fig = plt.gcf()
fig.set_size_inches(20,13)

plt.subplot(234)
sns.distplot(bank_df['euribor3m'])
fig = plt.gcf()
fig.set_size_inches(20,13)

plt.subplot(235)
sns.distplot(bank_df['nr_employed'])
fig = plt.gcf()
fig.set_size_inches(20,13)
```

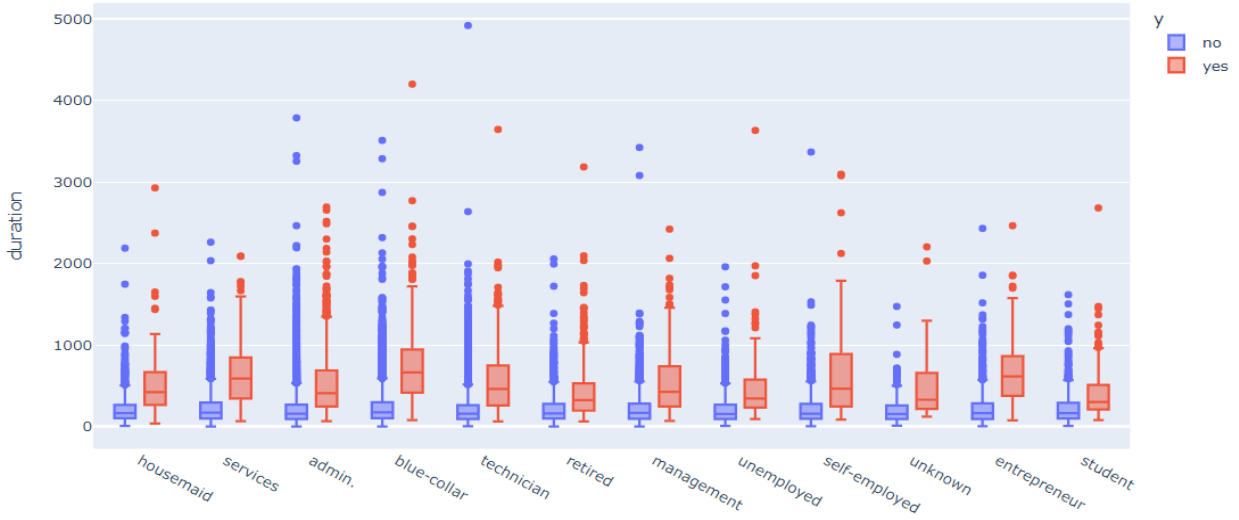


From these graphs, we can see that the numerical variables are not normally distributed. However, the analytical methods that we will be using will not be affected by this, so we are free to move on to the next step.

After univariate analysis, we will conduct some bivariate analysis to look at the interactions between some multiple variables. First, we will examine how the response variable changes between various values of job and duration:

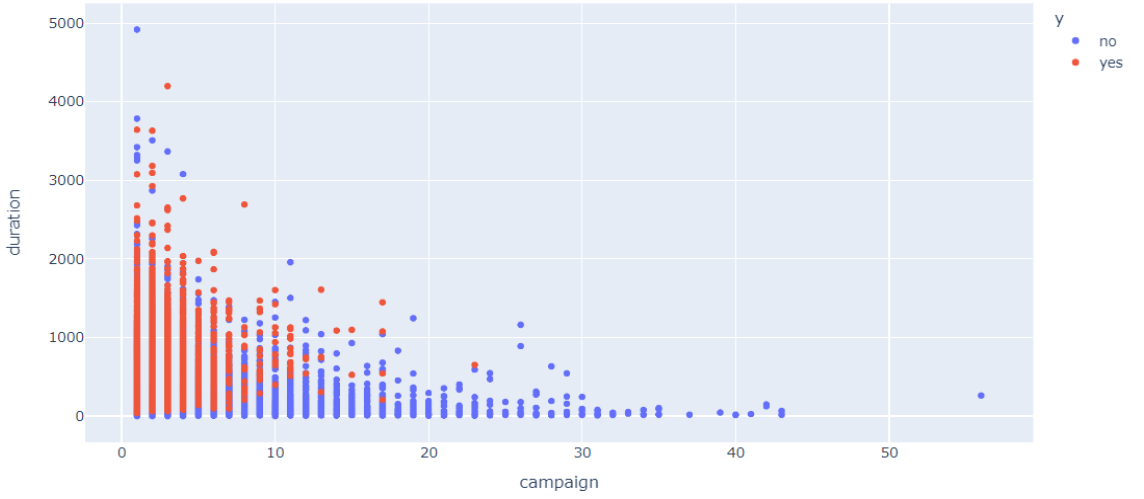
Bivariate Analysis

```
In [17]: import plotly.express as px

fig = px.box(bank_df, x="job", y="duration", color="y")
fig.update_traces(quartilemethod="exclusive") # or "inclusive", or "linear" by default
fig.show()
```



We also will look at how the response variable changes for different values of duration throughout the campaign:

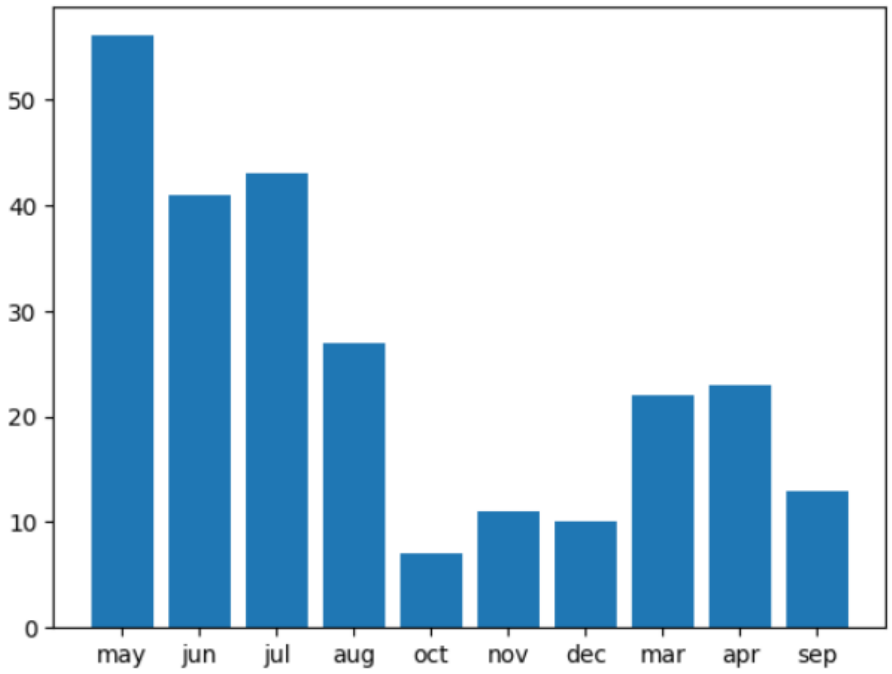
```
In [18]: fig = px.scatter(bank_df, x="campaign", y="duration", color="y")
fig.show()
```



We can see that calls with a higher duration were more likely to result in a response of yes and a deposit being made. Also, we see that more of these long calls were made early on in the campaign, and more successful calls were also made early in the campaign.

In addition, we can see how the values of the campaign variable are distributed across different months:

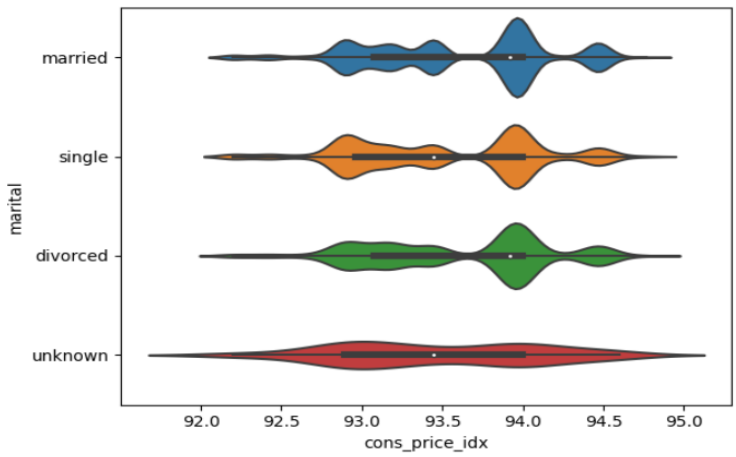
```
In [19]: plt.bar(bank_df['month'], bank_df['campaign'])
Out[19]: <BarContainer object of 41188 artists>
```



Here, we can see that much of the campaign was occurred in May, June, and July. We can interpret that these summer months bear less of a financial burden, so customers are more willing to deposit spare funds that they have during this time.

Finally, we will look at how marital status affects the cons_price_index variable:

```
In [20]: sns.violinplot( y=bank_df["marital"], x=bank_df["cons_price_idx"])
Out[20]: <AxesSubplot: xlabel='cons_price_idx', ylabel='marital'>
```



We see that those who are married have the highest average cons_price_index. These individuals are likely to have a higher CPI because they are more likely to be purchasing goods for a spouse and children than customers of other marital statuses.

After conducting these univariate and bivariate analyses, we moved on to more advanced analytical methods. First, we conducted K-Means clustering with 2 clusters to group the observations into “yes” and “no” groups. We also performed five different supervised learning methods: Logistic Regression, Decision Tree, Random Forest, Gradient-Boosted Tree, and Linear SVM. We found that the Logistic Regression, Gradient-Boosted Tree, and Linear SVM all performed well, but the best-performing model was the Gradient-Boosted Tree. The training and evaluation codes for these models can be seen in the included notebook.

We attempted to export a file for this LSVM model, but we encountered errors relating to the pickling process. When attempting to use the “.save(path)” method, we encountered an error stating that “path should be a string, got type %s”

```
gbtModel.save("D:/Libraries/Documents")
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
Cell In [78], line 3
      1 # gbtModel = gbt.fit(train)
      2 # gbtModel.save("D:/Libraries/Documents")
----> 3 gbtModel.save("D:/Libraries/Documents")

File C:\Apps\spark-3.3.0-bin-hadoop3\python\pyspark\ml\util.py:246, in MLWritable.save(self, path)
    244 def save(self, path: str) -> None:
    245     """Save this ML instance to the given path, a shortcut of 'write().save(path)'."""
--> 246     self.write().save(path)

File C:\Apps\spark-3.3.0-bin-hadoop3\python\pyspark\ml\util.py:197, in JavaMLWriter.save(self, path)
    195 if not isinstance(path, str):
    196     raise TypeError("path should be a string, got type %s" % type(path))
--> 197 self._jwrite.save(path)
```

We attempted to use the “.save(sparkContext,path)” method instead, but we encountered an error stating that “save() takes 2 positional arguments but 3 were given”

```
gbtModel.save(sc,"D:/Libraries/Documents")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [79], line 3
      1 # gbtModel = gbt.fit(train)
      2 # gbtModel.save("D:/Libraries/Documents")
----> 3 gbtModel.save(sc,"D:/Libraries/Documents")

TypeError: save() takes 2 positional arguments but 3 were given
```

We also tried using the joblib library and pickle library, but both times we received errors that "cannot pickle '_thread.RLock' object:

```
import joblib
```

```
fileName = "gbtModel.joblib"
joblib.dump(gbtModel, "./" + fileName)
```

```
--> 997             save(v)
    998     write(SETITEMS)
    999 elif n:

File c:\users\lotr_\appdata\local\programs\python\pythor
(self, obj)
    352     wrapper.write_array(obj, self)
    353     return
--> 355 return Pickler.save(self, obj)

File c:\users\lotr_\appdata\local\programs\python\pythor
    576 reduce = getattr(obj, "__reduce_ex__", None)
    577 if reduce is not None:
--> 578     rv = reduce(self.proto)
    579 else:
    580     reduce = getattr(obj, "__reduce__", None)

TypeError: cannot pickle '_thread.RLock' object
```

```
import pickle
```

```
pickle.dump(gbtModel, open('D:/Libraries/Documents/gbtModel.pkl', 'wb'))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [83], line 1
----> 1 pickle.dump(gbtModel, open('D:/Libraries/Documents/gbtModel.pkl', 'wb'))

TypeError: cannot pickle '_thread.RLock' object
```

Because of these errors, we were unable to export a serialized file of our best model. However, the model should be able to be reproduced using the attached notebook.