eda

December 7, 2025

1 EDA Notebook - Exploring bmarket dataset

1.1 ## Part 1 - Data Loading and initial EDA (Darren's)

Purpose - Convert data to a pandas DataFrame - Identify features and target column - Get a feel for the structure of the data before column specific EDA

---

[1]:
```
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.impute import KNNImputer
```

[2]:
```
import sqlite3

conn = sqlite3.connect("data/01_raw/bmarket.db")
cursor = conn.cursor()

# sqlite_master exists in all sqlite databases and contains metadata on all␣
 ↪tables
cursor.execute("SELECT name FROM sqlite_master")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

```
('bank_marketing',)
```

[3]:
```
import pandas as pd

# There's only a singular table, convert it to DataFrame
# From the head, it looks like Subscription Status is the target column
df = pd.read_sql("SELECT * FROM bank_marketing", conn)
df.head()
```

[3]:

	Client ID	Age	Occupation	Marital Status	Education Level	\
0	32885	57 years	technician	married	high.school	

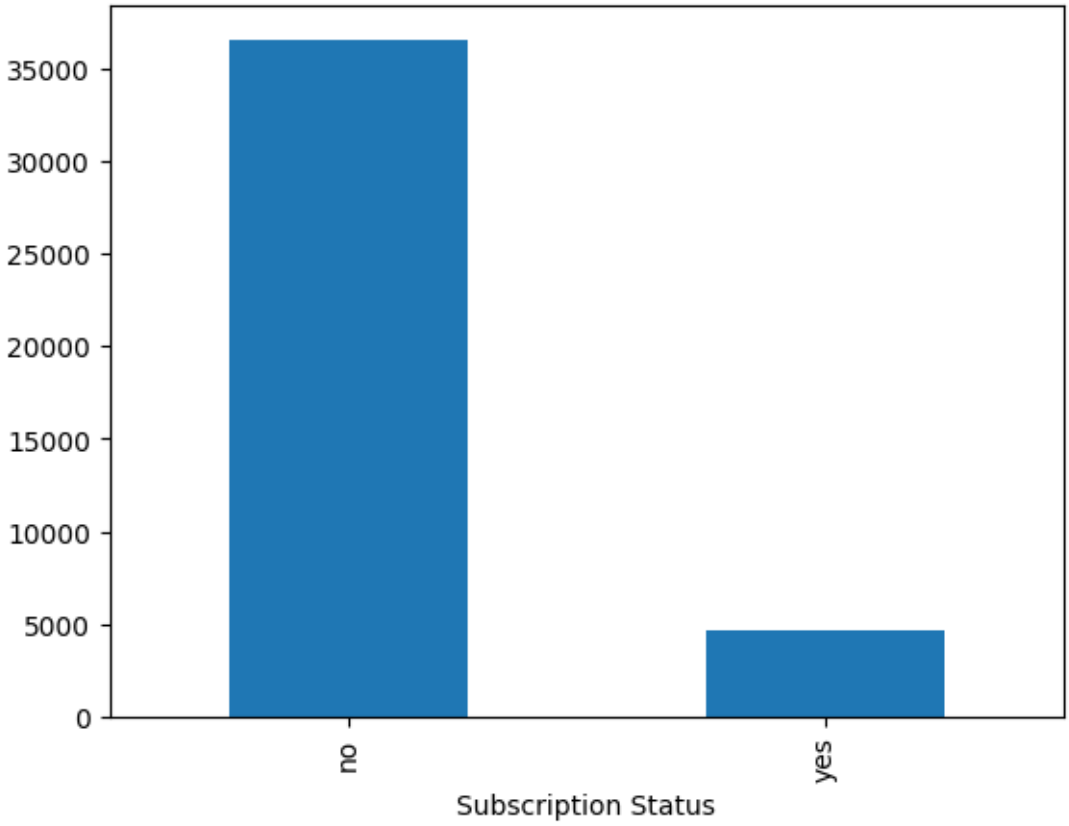
```
1       3170  55 years      unknown        married         unknown
2      32207  33 years  blue-collar        married        basic.9y
3       9404  36 years       admin.        married     high.school
4      14021  27 years    housemaid        married     high.school

  Credit Default Housing Loan Personal Loan Contact Method  Campaign Calls  \
0             no           no           yes           Cell               1
1        unknown          yes            no      telephone               2
2             no           no            no       cellular               1
3             no           no            no      Telephone               4
4             no         None            no           Cell               2

   Previous Contact Days Subscription Status
0                    999                  no
1                    999                  no
2                    999                  no
3                    999                  no
4                    999                  no
```

[4]:
```
# Null values only in two columns, but there is a significant amount in the housing_loan column
# age should be converted to integer type in cleaning
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Client ID             41188 non-null  int64
 1   Age                   41188 non-null  object
 2   Occupation            41188 non-null  object
 3   Marital Status        41188 non-null  object
 4   Education Level       41188 non-null  object
 5   Credit Default        41188 non-null  object
 6   Housing Loan          16399 non-null  object
 7   Personal Loan         37042 non-null  object
 8   Contact Method        41188 non-null  object
 9   Campaign Calls        41188 non-null  int64
 10  Previous Contact Days  41188 non-null  int64
 11  Subscription Status   41188 non-null  object
dtypes: int64(3), object(9)
memory usage: 3.8+ MB
```

[5]:
```
# Very heavy class imbalance
df['Subscription Status'].value_counts().plot(kind='bar')
df['Subscription Status'].value_counts()/df.shape[0]
```

[5]:
```
Subscription Status
no     0.887346
yes    0.112654
Name: count, dtype: float64
```



1.2 ## Part 2a - Data Cleaning: Darren's Part

Purpose - Go column-by-column to explore and understand each feature - Determine all cleaning to be performed in the Kedro pipeline

---

1.2.1 Initial Simple Data Cleaning

[6]:
```
# Renaming all columns to a format without spaces (for easier usage in pandas)
column_renames = {name : name.lower().replace(" ", "_") for name in df.columns}
df_renamed = df.rename(columns=column_renames)
df_renamed.columns
```

[6]:
```
Index(['client_id', 'age', 'occupation', 'marital_status', 'education_level',
       'credit_default', 'housing_loan', 'personal_loan', 'contact_method',
```

```
       'campaign_calls', 'previous_contact_days', 'subscription_status'],
      dtype='object')
```

[7]:
```
# Dropping the Client ID column
# This data is not needed because ID should not have any non-coincidental␣
 ↪correlation with subscription status
df_dropid = df_renamed.drop("client_id", axis=1)
df_dropid.head(3)
```

[7]:

	age	occupation	marital_status	education_level	credit_default	housing_loan	personal_loan	contact_method	campaign_calls	previous_contact_days	subscription_status
0	57 years	technician	married	high.school	no	no	yes	Cell	1	999	no
1	55 years	unknown	married	unknown	unknown	yes	no	telephone	2	999	no
2	33 years	blue-collar	married	basic.9y	no	no	no	cellular	1	999	no

1.3 ### Age

Unrealistic 150 year old outliers (likely data entry errors) found - Converted column to integer type and stripped 'years' unit - Set to -1 and added an extra column denoting if the age was unknown or not (boolean) —

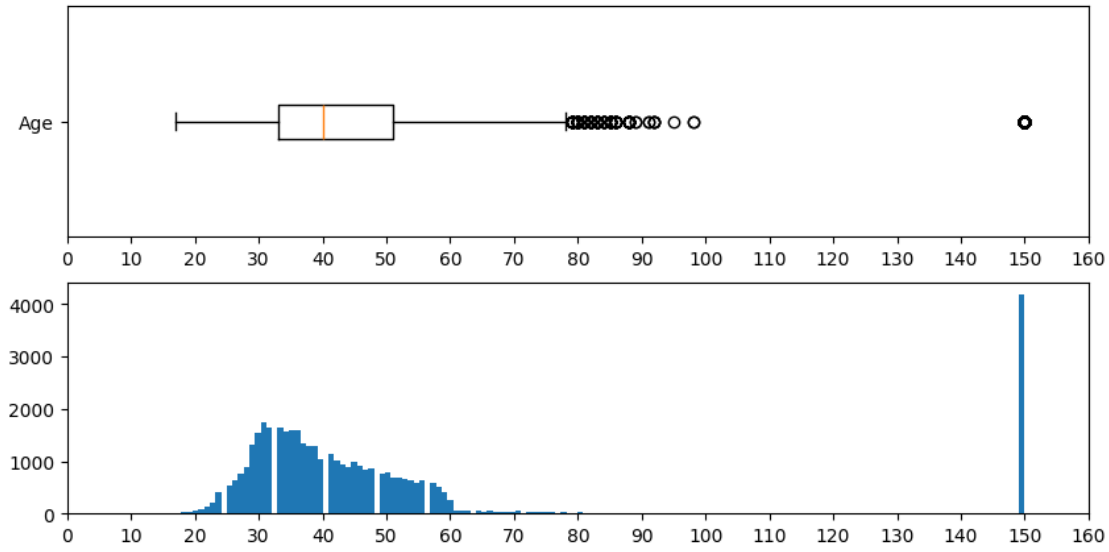
[8]:
```
# AGE
# Conversion from string to int type
df_age = df_dropid.copy()
df_age.age = df_age.age.map(lambda x: int(x.split(" ")[0]))

# Checking for any empty age cells
print(f"Number of null values in age column: {df_age.age.isna().sum()}")
df_age.dtypes
```

```
Number of null values in age column: 0
```

[8]:
```
age                    int64
occupation            object
marital_status        object
education_level       object
credit_default        object
housing_loan          object
personal_loan         object
```

```
contact_method           object
campaign_calls            int64
previous_contact_days     int64
subscription_status      object
dtype: object
```

[9]:
```
# Plotting the age column to show distribution (boxplot and histogram)
fig, ax = plt.subplots(2, 1, figsize=(10, 5))
ax[0].boxplot(df_age.age, vert=False, tick_labels=["Age"])
ax[0].set_xticks(range(0, 161, 10))
ax[1].hist(df_age.age, bins=150)
ax[1].set_xticks(range(0, 161, 10))
plt.show()
```



The frequency analysis of the age column shows a large number of outliers with age = 150, this is likely a data entry error

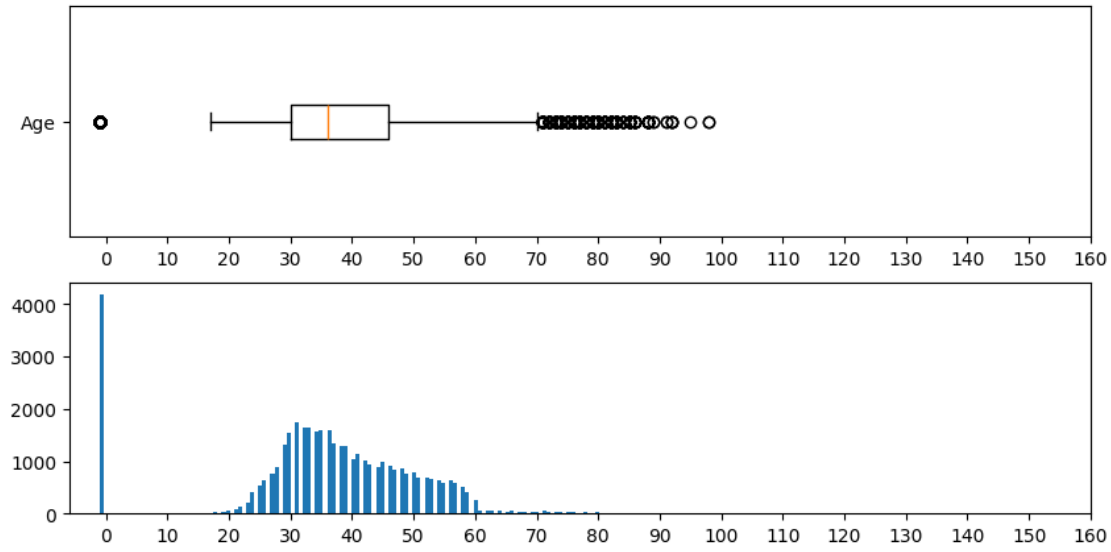
For now there are 2 possible solutions: 1. Clip to max age (~90 years old) 2. Set all outliers as -1 to denote unknown

In this case, it would be best to set them as unknown, since clipping the values would skew the dataset toward the max age, and ruin the distribution of ages in this dataset. Most likely making the model less accurate.

Solution: Set all ages = 150 (outliers) to -1, Create a new boolean column denoting that the age is unknown for extra information to the model later on.

[10]:
```
# Convert all 150 year olds to -1 (unknown)
df_age.age = df_age.age.map(lambda x: x if x != 150 else -1)
```

```
df_age.insert(loc=df_age.columns.get_loc("age")+1, column="age_unk",␣
 ↪value=df_age.age.map(lambda x: True if x == -1 else False))

# Showing new frequency graph
fig, ax = plt.subplots(2, 1, figsize=(10, 5))
ax[0].boxplot(df_age.age, vert=False, tick_labels=["Age"])
ax[0].set_xticks(range(0, 161, 10))
ax[1].hist(df_age.age, bins=150)
ax[1].set_xticks(range(0, 161, 10))
plt.show()

df_age.loc[:, ["age", "age_unk"]].head()
```



[10]:

	age	age_unk
0	57	False
1	55	False
2	33	False
3	36	False
4	27	False

Summary 1st problem, age column has dtype of object, meaning that it has string values. This can be solved by converting the string data values in the age column e.g. “28 years” –> 28 (int).

2nd problem, there are many outliers in the dataset (150 year olds). To deal with them, I convert them to -1 and denote them as unknown in a seperate column.

1.4 ### Occupation

Normal, no cleaning required

[11]:
```
df_job = df_age.copy()
df_job.head()
```

[11]:

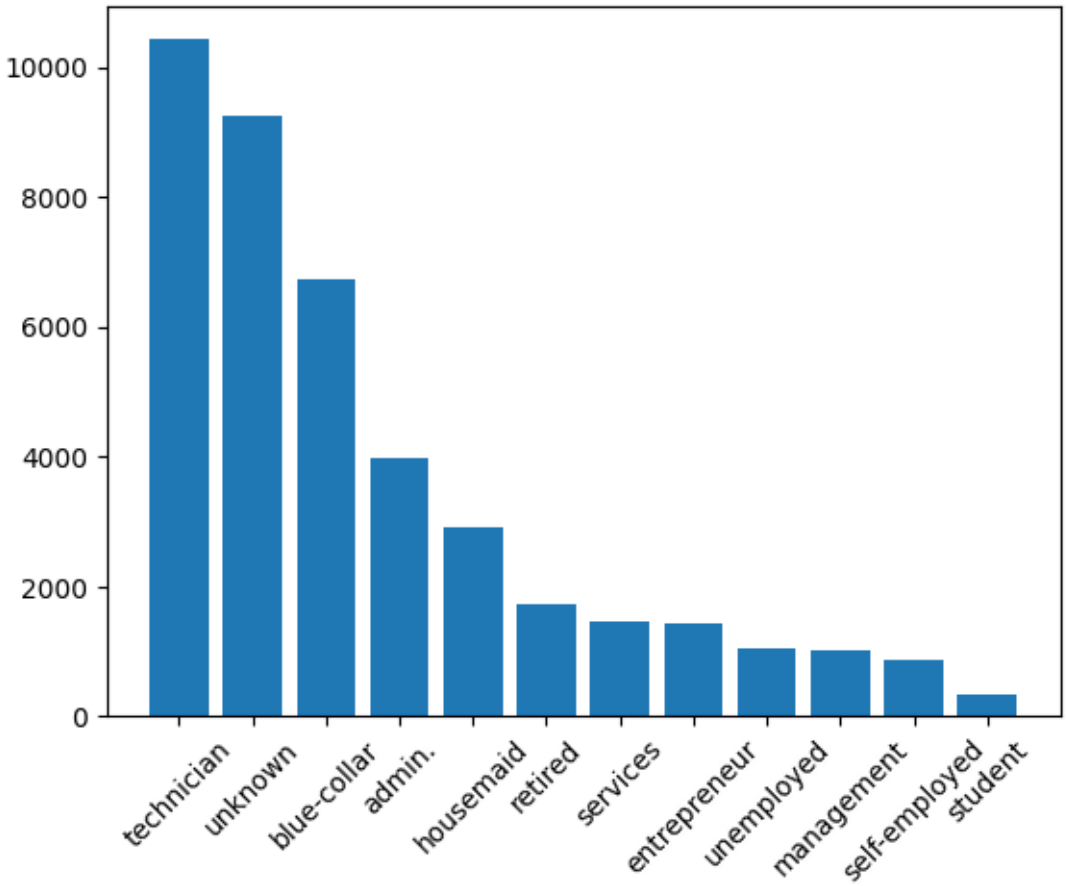
	age	age_unk	occupation	marital_status	education_level	credit_default
0	57	False	technician	married	high.school	no
1	55	False	unknown	married	unknown	unknown
2	33	False	blue-collar	married	basic.9y	no
3	36	False	admin.	married	high.school	no
4	27	False	housemaid	married	high.school	no

	housing_loan	personal_loan	contact_method	campaign_calls
0	no	yes	Cell	1
1	yes	no	telephone	2
2	no	no	cellular	1
3	no	no	Telephone	4
4	None	no	Cell	2

	previous_contact_days	subscription_status
0	999	no
1	999	no
2	999	no
3	999	no
4	999	no

[12]:
```
print(f"Number of null values in occupation column: {df_job.occupation.isna().
  ↪sum()}")
print(df_job.occupation.value_counts())
plt.bar(df_job.occupation.unique(), df_job.occupation.value_counts())
plt.xticks(rotation=45)
plt.show()
```

```
Number of null values in occupation column: 0
occupation
admin.           10422
blue-collar       9254
technician        6743
services          3969
management        2924
retired           1720
entrepreneur      1456
self-employed     1421
housemaid         1060
unemployed        1014
student            875
unknown            330
Name: count, dtype: int64
```



Summary For the occupation column, there are no big problems. There are no null values in this column.

There is one tiny problem, which is the "admin." category having the extra "." in the name, but this does not really need to be fixed here.

1.5 ### Education Level

Normal, no cleaning required

Rare category: illiterate, with only 18 instances in entire dataset

---

[13]:
```
df_education = df_job.copy()
df_education.head()
```

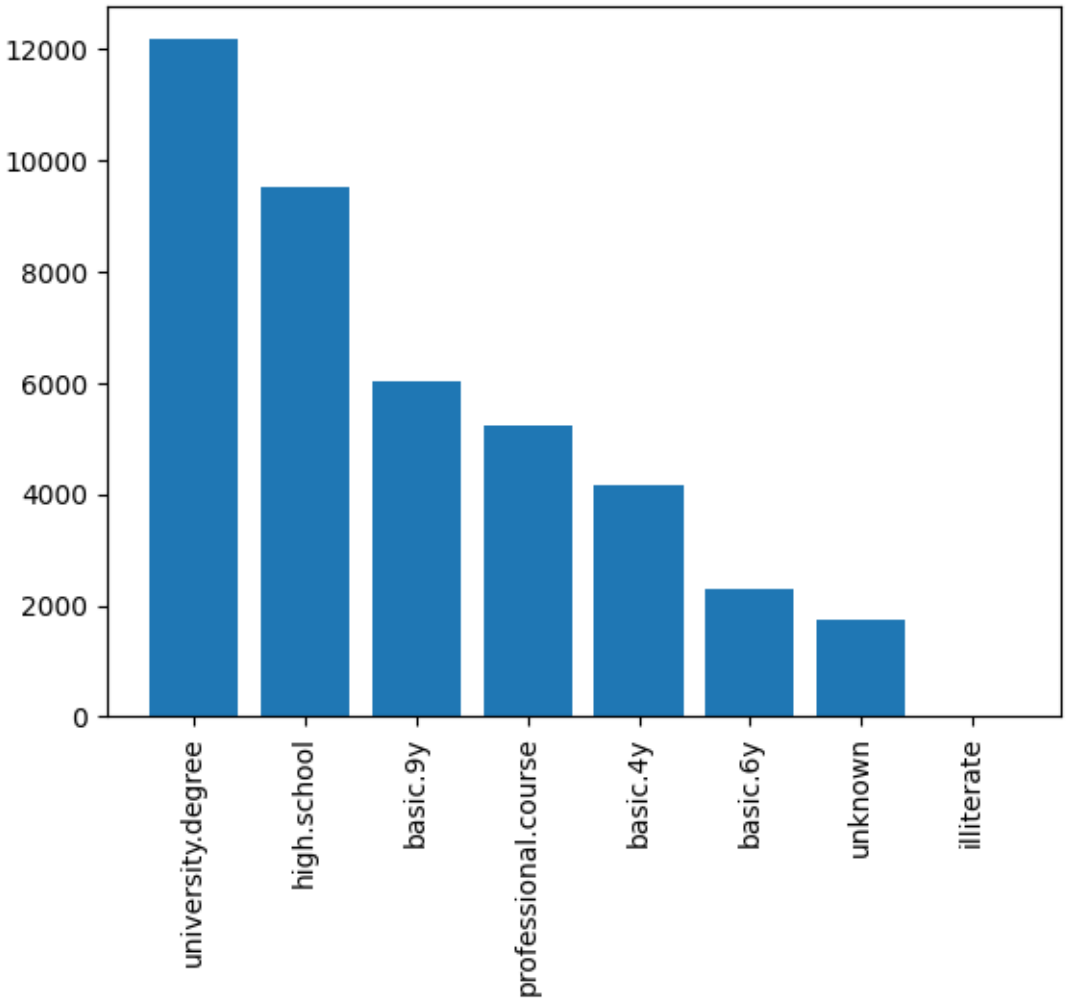
[13]:

	age	age_unk	occupation	marital_status	education_level	credit_default	\
0	57	False	technician	married	high.school	no	
1	55	False	unknown	married	unknown	unknown	

```
2   33    False  blue-collar        married        basic.9y             no
3   36    False       admin.        married     high.school             no
4   27    False    housemaid        married     high.school             no

  housing_loan personal_loan contact_method  campaign_calls  \
0           no           yes           Cell               1
1          yes            no      telephone               2
2           no            no       cellular               1
3           no            no      Telephone               4
4         None            no           Cell               2

   previous_contact_days subscription_status
0                    999                  no
1                    999                  no
2                    999                  no
3                    999                  no
4                    999                  no
```

[14]:
```
# Unique values of education level show no overlap in categorization, so no change is necessary
# No nulls found
# Small problem with the extremely small amount of data on illiterate education level, but otherwise there is nothing serious that needs to be cleaned.
print(f"Number of null values in education_level column: {df_education.education_level.isna().sum()}")
print(df_education.education_level.value_counts())
plt.bar(df_education.education_level.value_counts().index, df_education.education_level.value_counts().values)
plt.xticks(rotation=90)
plt.show()
```

```
Number of null values in education_level column: 0
education_level
university.degree      12168
high.school             9515
basic.9y                6045
professional.course     5243
basic.4y                4176
basic.6y                2292
unknown                 1731
illiterate                18
Name: count, dtype: int64
```



Summary The education_level column also does not have any big problems with its data.

The only thing of note is the "illiterate" category, which only has 18 samples in the entire dataset.

1.6 ### Contact Method

Categorization overlap: Cell = cellular & Telephone = telephone - Combined categories (Cell & cellular -> cellular, Telephone & telephone -> telephone)

---

[15]:
```
df_contact = df_education.copy()
df_contact.head()
```

[15]:
```
   age  age_unk   occupation marital_status education_level credit_default  \
0   57    False   technician        married     high.school             no
1   55    False      unknown        married         unknown        unknown
```

```
2   33    False  blue-collar        married        basic.9y             no
3   36    False       admin.        married     high.school             no
4   27    False    housemaid        married     high.school             no

  housing_loan personal_loan contact_method  campaign_calls  \
0           no           yes           Cell               1
1          yes            no      telephone               2
2           no            no       cellular               1
3           no            no      Telephone               4
4         None            no           Cell               2

   previous_contact_days subscription_status
0                    999                  no
1                    999                  no
2                    999                  no
3                    999                  no
4                    999                  no
```

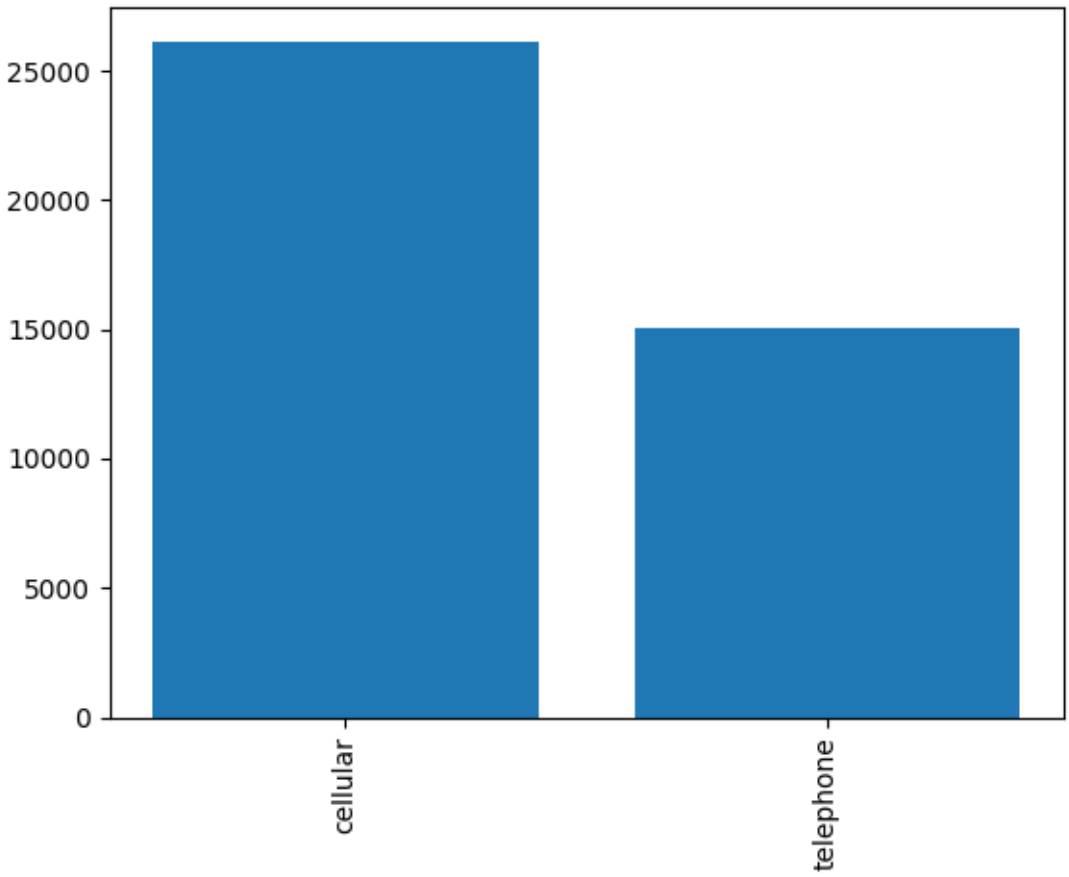
[16]:
```python
# No nulls found
# Categorization overlap with Cell -> cellular & Telephone -> telephone
print(f"Number of null values in contact_method column: {df_contact.
 ↪contact_method.isna().sum()}")
print(_ := df_contact.contact_method.value_counts())
print(f"\nCellular Category total: {_.iloc[:2].sum()}\nTelephone Category total:
 ↪ {_.iloc[2:].sum()}")
```

```
Number of null values in contact_method column: 0
contact_method
Cell         13100
cellular     13044
Telephone     7585
telephone     7459
Name: count, dtype: int64

Cellular Category total: 26144
Telephone Category total: 15044
```

[17]:
```python
# Combining Cell -> cellular & Telephone -> telephone
df_contact.contact_method = df_contact.contact_method.map(lambda x: "cellular"␣
 ↪if x[0].lower() == "c" else "telephone")
print(df_contact.contact_method.value_counts())
```

```
contact_method
cellular     26144
telephone    15044
Name: count, dtype: int64
```

[18]:
```
plt.bar(df_contact.contact_method.unique(), df_contact.contact_method.
  ↪value_counts())
plt.xticks(rotation=90)
plt.show()
```



Summary The contact_method has 1 major problem, that being the category overlap of Cell -> cellular and Telephone -> telephone.

This can be easily solved by merging the similar categories into one (cellular & telephone)

1.7 ### Campaign Calls

Campaign Calls data sometimes represented with negative numbers (not possible) - Used K-S test to prove distribution similarity between negative and positive space numbers - Assume negative numbers is a data entry error where a proportion of positive numbers were converted by accident - Abs on Campaign Calls columns

---

[19]:
```
df_campaign = df_contact.copy()
df_campaign.head()
```

[19]:

	age	age_unk	occupation	marital_status	education_level	credit_default
0	57	False	technician	married	high.school	no
1	55	False	unknown	married	unknown	unknown
2	33	False	blue-collar	married	basic.9y	no
3	36	False	admin.	married	high.school	no
4	27	False	housemaid	married	high.school	no

	housing_loan	personal_loan	contact_method	campaign_calls
0	no	yes	cellular	1
1	yes	no	telephone	2
2	no	no	cellular	1
3	no	no	telephone	4
4	None	no	cellular	2

	previous_contact_days	subscription_status
0	999	no
1	999	no
2	999	no
3	999	no
4	999	no

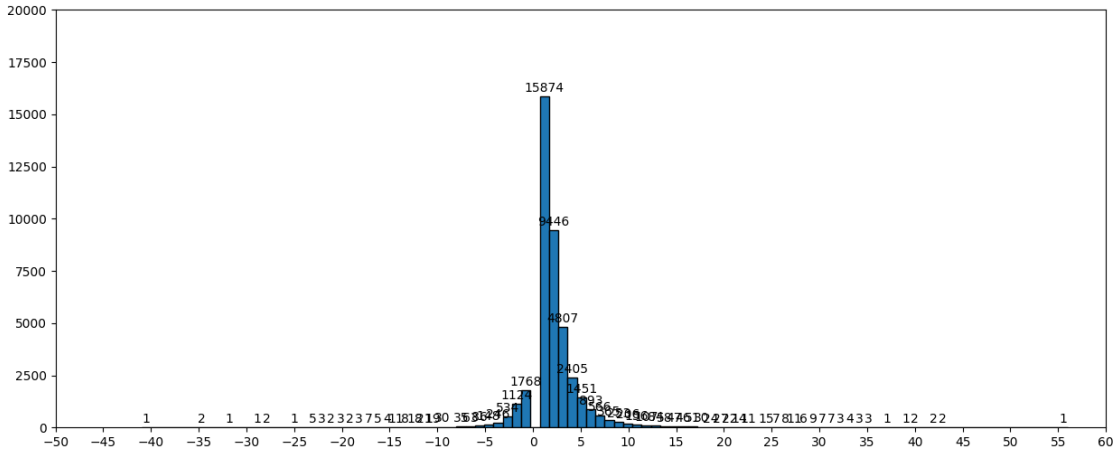
[20]:
```
# No null values
# It looks like campaign calls goes into the negatives, despite it not being possible to have negative campaign calls
print(f"Number of null values in contact_method column: {df_contact.contact_method.isna().sum()}")
df_campaign.campaign_calls.describe()
```

```
Number of null values in contact_method column: 0
```

[20]:
```
count    41188.000000
mean         2.051374
std          3.171345
min        -41.000000
25%          1.000000
50%          2.000000
75%          3.000000
max         56.000000
Name: campaign_calls, dtype: float64
```

[21]:
```
plt.figure(figsize=(15, 6))
counts, bins, patches = plt.hist(df_campaign.campaign_calls, bins=100, edgecolor='black')
# Add value labels
for count, bin_edge in zip(counts, bins[:-1]):
```

```
    if count > 0:
        plt.text(bin_edge + (bins[1] - bins[0]) / 2, count+100,␣
 ↪str(int(count)), ha='center', va='bottom')
plt.xticks(range(-50, 61, 5))
plt.axis([-50, 60, 0, 20000])
plt.show()
```



[22]:
```
print("Count of negative values: ", end="")
print(len(df_campaign[df_campaign.campaign_calls < 0]))
```

```
Count of negative values: 4153
```

Based on the histogram, you can see many of the values are in the negatives, despite these values not being possible for this data column.

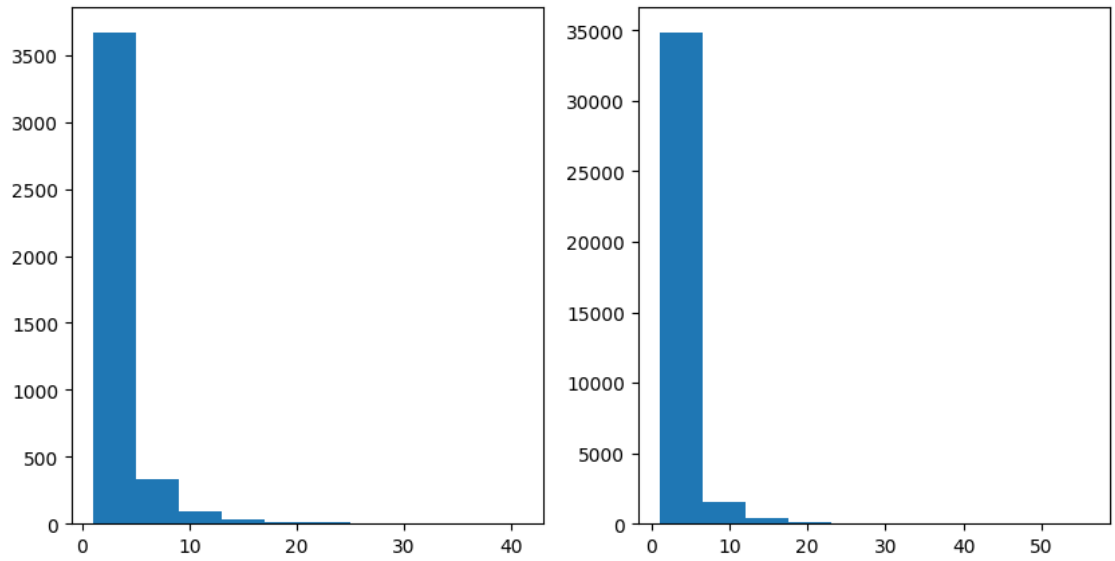
3 Solutions: 1. Clip negative values to 0 / lowest campaign calls values > 0 2. Change all negative values into unknown values 3. Assume this is a data entry error where positive data was mistakenly changed to negative values

Solution 1: - Clipping the negative values will create skewed data, since the histogram already shows that the data is more frequent the smaller the number is. Additional ~4,000 data points would be added to the most frequent category, which will cause the distribution after cleaning to not be able to closely replicate the original distribution.

Solution 2: - Changing all negative values into unknown values will result in losing ~4,000 potential data points, which can affect the model's ability

Solution 3: - Based on the histogram above, it does seem that a portion of the positive values were accidentally converted into negative values, since the overall distribution in the negative space is similar to that of the positive space. Additionally, converting the data this way would allow us to prevent any data loss which would have been caused by the previous solutions. The only problem is proving that the distributions of the positive and negative space of the campaign_calls column are similar enough that its possible they are from the same distribution.

[23]:
```
# Plotting the negative and positive sample spaces of the column
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
# Negative space
ax[0].hist(neg := df_campaign.campaign_calls[df_campaign.campaign_calls < 0].
 ↪abs())
# Positive space
ax[1].hist(pos := df_campaign.campaign_calls[df_campaign.campaign_calls > 0])
plt.show()
```



[24]:
```
# Calculating similarity using 2 sample K-S test
D_stat, p = stats.ks_2samp(neg, pos)
print("K-S Test for samples neg & pos:")
print(f"D Stat:{D_stat:>25}")
print(f"p-value:{p:>24}")
```

```
K-S Test for samples neg & pos:
D Stat:     0.012686471895130724
p-value:      0.5804227545723576
```

Explanation & Thought Process:

The hypothesis here is that the negative numbers inside the dataset (negative sample), are errors that are caused by accidentally converting a portion of the positive numbers into negative numbers. This hypothesis can be proven if both the negative and positive samples are likely taken from the same distribution. In order to do this, I will use the 2 sample Kolmogorov-Smirnov (K-S) Test which is used to determine if two samples come from the same distribution. This will be calculated using scipy.stats.ks_2samp()

The Kolmogorov-Smirnov (KS) test defines two hypotheses: Null Hypothesis: Samples are taken from the same distribution Alternative Hypothesis: Samples are taken from different distributions

If the p-value <= alpha (0.05, the standard significance level) the null hypothesis is rejected and the samples have different distributions. If the p-value > alpha, the null hypothesis cannot be rejected and it is likely that the two samples come from the same distribution. Additionally, the D statistic shows the maximium distance between the Cumulative Density Function (CDF) curves. A smaller D statistic and higher p-value typically suggests similar distributions.

Application:

I separated the campaign_calls column into two separate samples, one which contains only the positive values and one that only contains the negative values absoluted to be put in the positive space. Then I calculated the D statistic and p-value using 2 sample K-S test to determine the distribution similarity.

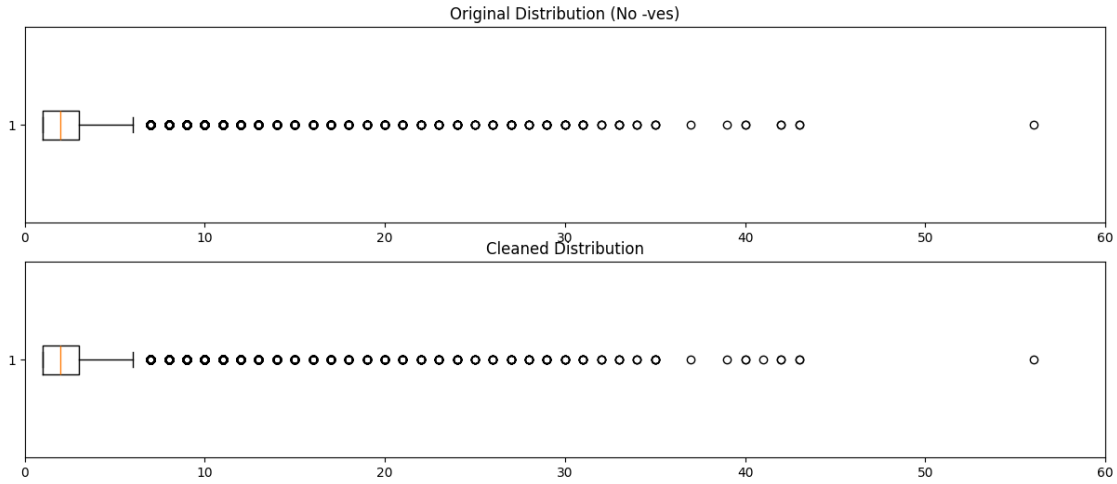
As a result, I obtained the p-value of ~0.5804 which is larger than the alpha / standard significance level of 0.05. Which allows me to reject the null hypothesis. I also obtained a D statistic of ~0.01268. With both the low D statistic and the high p-value it is safe to say that the negative space and positive space campaign_calls data values are likely taken from the same distribution.

[25]:
```
# Capture the original data before cleaning
print("Original Data (No -ves)")
print(df_campaign.loc[df_campaign.campaign_calls > 0].campaign_calls.describe())
fig, ax = plt.subplots(2, 1, figsize=(15, 6))
ax[0].boxplot(df_campaign.loc[df_campaign.campaign_calls > 0].campaign_calls,␣
 ↪vert=False)
ax[0].set_title("Original Distribution (No -ves)")

# Converting all numbers to positive via abs(), following solution #3
df_campaign.campaign_calls = df_campaign.campaign_calls.abs()

# Capture the new cleaned data distribution
print("\nCleaned Data")
print(df_campaign.campaign_calls.describe())
ax[1].boxplot(df_campaign.campaign_calls, vert=False)
ax[1].set_title("Cleaned Distribution")

for x in ax:
    x.set_xlim(0, 60)
plt.plot()
```

```
Original Data (No -ves)
count    37035.000000
mean         2.568462
std          2.763739
min          1.000000
25%          1.000000
50%          2.000000
75%          3.000000
max         56.000000
Name: campaign_calls, dtype: float64
```

```
Cleaned Data
count    41188.000000
mean         2.567593
std          2.770014
min          1.000000
25%          1.000000
50%          2.000000
75%          3.000000
max         56.000000
Name: campaign_calls, dtype: float64
```

[25]: []



Summary The campaign_calls column has negative values, which is not possible since you can only have a positive amount of calls. Seeing the similar distribution between the positive and negative samples spaces, I use the 2 sample K-S test to determine if the positive and negative space both come from the same distribution.

Since the 2 sample K-S test returned a low p-value, I could **not** reject the null hypothesis that the data came from the same distribution. Along with the low D-statistic, we can safely assume that both samples come from the same distribution.

As seen in the above plots, after cleaning the campaign_calls column, the distribution is replicated very well.

1.8 Part 2b - Data Cleaning: Harish's Part

1.9 ### Marital Status

Normal, no cleaning required

[26]:
```
df_marital = df_campaign.copy()
df_marital.head()
```

[26]:

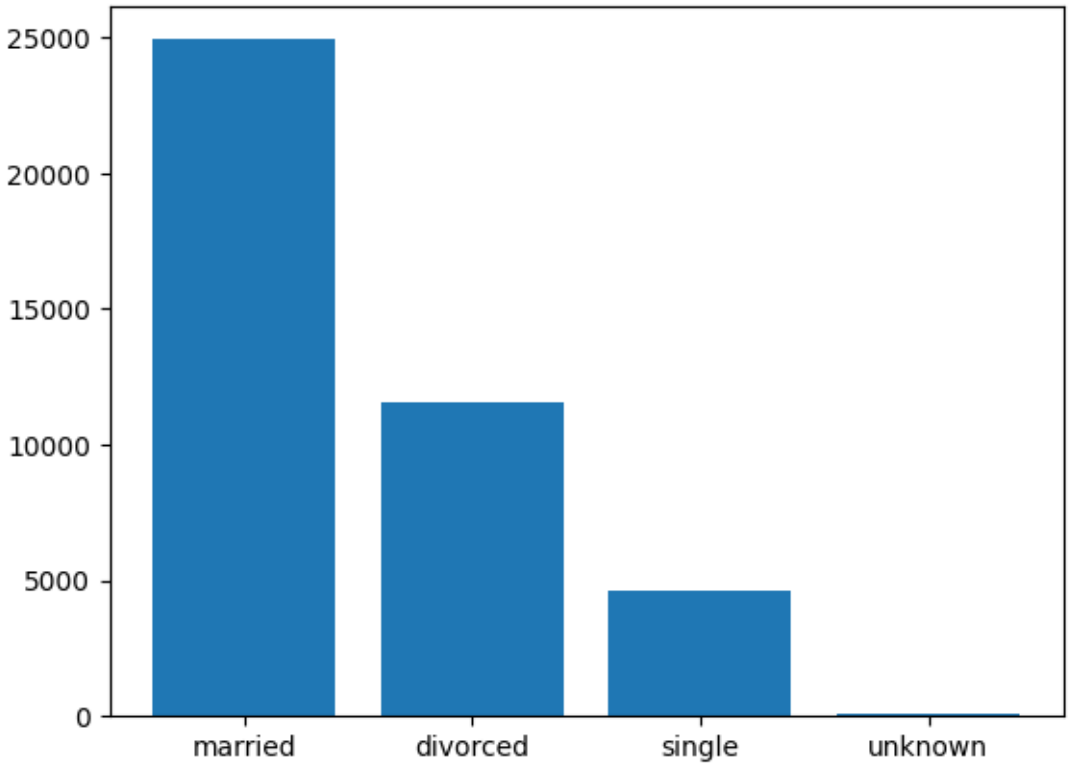
	age	age_unk	occupation	marital_status	education_level	credit_default
0	57	False	technician	married	high.school	no
1	55	False	unknown	married	unknown	unknown
2	33	False	blue-collar	married	basic.9y	no
3	36	False	admin.	married	high.school	no
4	27	False	housemaid	married	high.school	no

	housing_loan	personal_loan	contact_method	campaign_calls
0	no	yes	cellular	1
1	yes	no	telephone	2
2	no	no	cellular	1
3	no	no	telephone	4
4	None	no	cellular	2

	previous_contact_days	subscription_status
0	999	no
1	999	no
2	999	no
3	999	no
4	999	no

[27]:
```
# Unique values of marital status show no overlap in categorization, so no change is necessary
# No nulls in column
# Normal OHE for this column in pipeline
print(f"Number of null values in marital_status column: {df_marital.marital_status.isna().sum()}")
print(df_marital.marital_status.value_counts())
plt.bar(df_marital.marital_status.unique(), df_marital.marital_status.value_counts())
plt.show()
```

```
Number of null values in marital_status column: 0
marital_status
married     24928
single      11568
divorced     4612
unknown        80
Name: count, dtype: int64
```



Summary Normal column, low number of 'unknown' values could potentially be problematic but doesn't need to be addressed.

1.10 ### Previous Contact Days

Over 95% of values were set to 999 - Added new boolean column denoting 999 as 'not previously contacted' —

[28]:
```
# Model card mentions that 999 means no prior contact, identifying existence of␣
↪999 as a value
df_pdays = df_campaign.copy()
df_pdays.head()
```

[28]:
```
   age  age_unk   occupation marital_status education_level credit_default  \
0   57    False   technician        married     high.school             no
1   55    False      unknown        married         unknown        unknown
2   33    False  blue-collar        married       basic.9y             no
3   36    False       admin.        married     high.school             no
4   27    False    housemaid        married     high.school             no

  housing_loan personal_loan contact_method  campaign_calls  \
0           no           yes       cellular               1
```

```
1          yes            no      telephone               2
2           no            no       cellular               1
3           no            no      telephone               4
4         None            no       cellular               2

   previous_contact_days subscription_status
0                    999                  no
1                    999                  no
2                    999                  no
3                    999                  no
4                    999                  no
```

[29]:
```
# All the rows in the head showed 999, need to check if there are normal values
df_pdays['previous_contact_days'].describe()
```

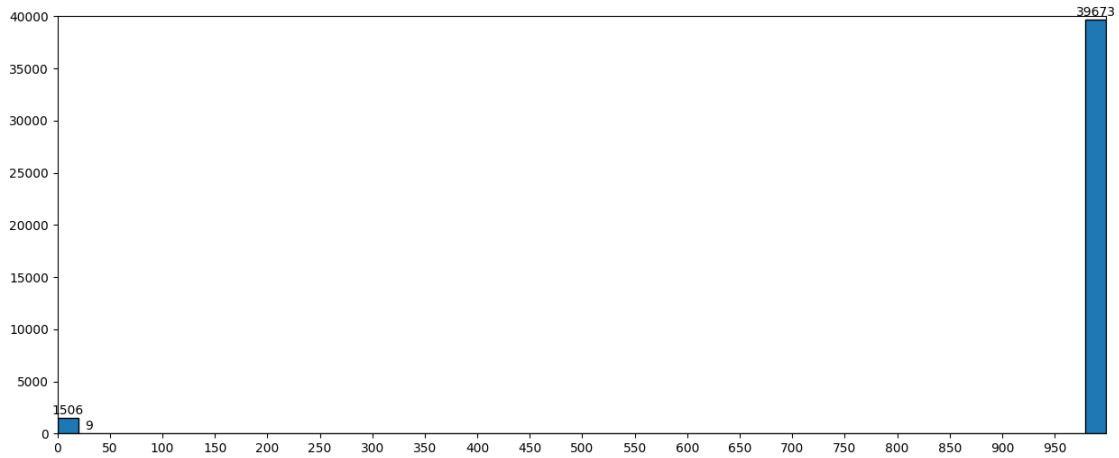
[29]:
```
count    41188.000000
mean       962.475454
std        186.910907
min          0.000000
25%        999.000000
50%        999.000000
75%        999.000000
max        999.000000
Name: previous_contact_days, dtype: float64
```

[30]:
```
# Normal values do exist, but it's a very small percentage
df_pdays[df_pdays['previous_contact_days'] < 999]['previous_contact_days'].
 ↪describe()
```

[30]:
```
count    1515.000000
mean        6.014521
std         3.824906
min         0.000000
25%         3.000000
50%         6.000000
75%         7.000000
max        27.000000
Name: previous_contact_days, dtype: float64
```

[31]:
```
plt.figure(figsize=(15, 6))
counts, bins, patches = plt.hist(df_pdays.previous_contact_days, bins=50,␣
 ↪edgecolor='black')
# Add value labels
for count, bin_edge in zip(counts, bins[:-1]):
    if count > 0:
        plt.text(bin_edge + (bins[1] - bins[0]) / 2, count+100,␣
 ↪str(int(count)), ha='center', va='bottom')
```

```
plt.xticks(range(0, 999, 50))
plt.axis([0, 999, 0, 40000])
plt.show()
```



[32]:
```
df_pdays.insert(loc=df_pdays.columns.get_loc("previous_contact_days")+1,␣
↪column="previously_contacted", value=df_pdays.previous_contact_days.
↪map(lambda x: False if x == 999 else True))
# Just to check whether correct booleans were added for values of below 999 and␣
↪999 respectively
df_pdays[df_pdays['previous_contact_days'] < 999].head()
```

[32]:

	age	age_unk	occupation	marital_status	education_level	credit_default	\
42	25	False	student	single	high.school	no	
53	24	False	admin.	married	basic.9y	no	
58	41	False	admin.	divorced	university.degree	no	
91	25	False	student	single	high.school	no	
137	31	False	admin.	married	university.degree	no	

	housing_loan	personal_loan	contact_method	campaign_calls	\
42	yes	no	cellular	1	
53	None	no	cellular	1	
58	yes	None	cellular	2	
91	yes	no	cellular	1	
137	None	no	cellular	2	

	previous_contact_days	previously_contacted	subscription_status
42	3	True	yes
53	10	True	yes
58	6	True	yes
91	6	True	no
137	13	True	yes

[33]:
```
# Make 999 into -1 to fix the bad skewing of the data and also make it obvious␣
 ↪it's not in normal range
df_pdays.previous_contact_days = df_pdays.previous_contact_days.apply(lambda x:␣
 ↪x if x < 999 else -1)
df_pdays[df_pdays['previous_contact_days'] == -1].head()
```

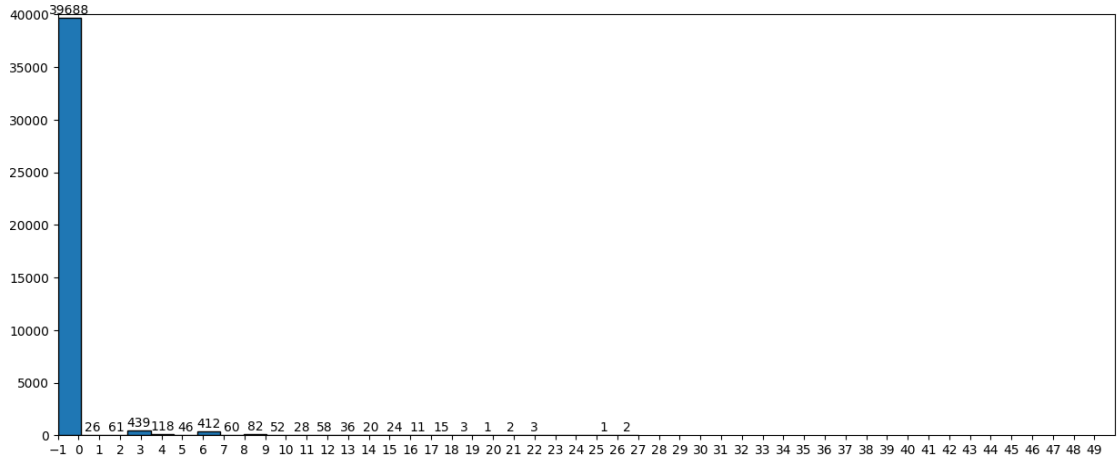
[33]:

	age	age_unk	occupation	marital_status	education_level	credit_default	\
0	57	False	technician	married	high.school	no	
1	55	False	unknown	married	unknown	unknown	
2	33	False	blue-collar	married	basic.9y	no	
3	36	False	admin.	married	high.school	no	
4	27	False	housemaid	married	high.school	no	

	housing_loan	personal_loan	contact_method	campaign_calls	\
0	no	yes	cellular	1	
1	yes	no	telephone	2	
2	no	no	cellular	1	
3	no	no	telephone	4	
4	None	no	cellular	2	

	previous_contact_days	previously_contacted	subscription_status
0	-1	False	no
1	-1	False	no
2	-1	False	no
3	-1	False	no
4	-1	False	no

[34]:
```
plt.figure(figsize=(15, 6))
counts, bins, patches = plt.hist(df_pdays.previous_contact_days, bins=25,␣
 ↪edgecolor='black')
# Add value labels
for count, bin_edge in zip(counts, bins[:-1]):
    if count > 0:
        plt.text(bin_edge + (bins[1] - bins[0]) / 2, count+100,␣
 ↪str(int(count)), ha='center', va='bottom')
plt.xticks(range(-1, 50))
plt.axis([-1, 50, 0, 40000])
plt.show()
```



Summary The dataset card explicitly states that a value of 999 means that the client was not previously contacted before, hence that is added as a new column. The value of 999 is changed to -1 to make it more obvious it is outside normal range.

1.11 ### Credit Default

Category of 'yes' only has three instances across all rows - 'yes' and 'no' category combined into 'known', 'unknown' remains the same —

[35]:
```
df_credit = df_pdays.copy()
df_credit.head()
```

[35]:

	age	age_unk	occupation	marital_status	education_level	credit_default	\
0	57	False	technician	married	high.school	no	
1	55	False	unknown	married	unknown	unknown	
2	33	False	blue-collar	married	basic.9y	no	
3	36	False	admin.	married	high.school	no	
4	27	False	housemaid	married	high.school	no	

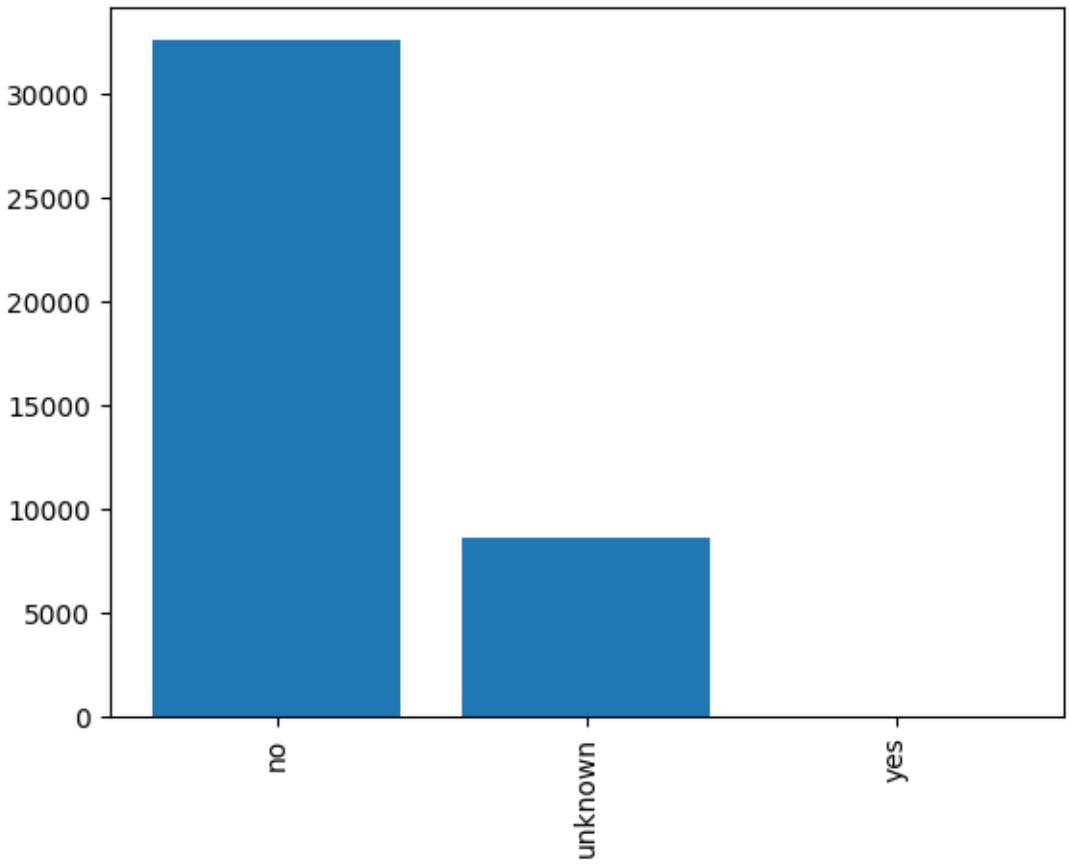
	housing_loan	personal_loan	contact_method	campaign_calls	\
0	no	yes	cellular	1	
1	yes	no	telephone	2	
2	no	no	cellular	1	
3	no	no	telephone	4	
4	None	no	cellular	2	

	previous_contact_days	previously_contacted	subscription_status
0	-1	False	no
1	-1	False	no
2	-1	False	no
3	-1	False	no

```
4                    -1                  False                    no
```

[36]:
```
# Extremely low number of yes values
print(df_credit.credit_default.value_counts())
plt.bar(df_credit.credit_default.value_counts().index, df_credit.credit_default.
  ↪value_counts().values)
plt.xticks(rotation=90)
plt.show()
```

```
credit_default
no         32588
unknown     8597
yes            3
Name: count, dtype: int64
```



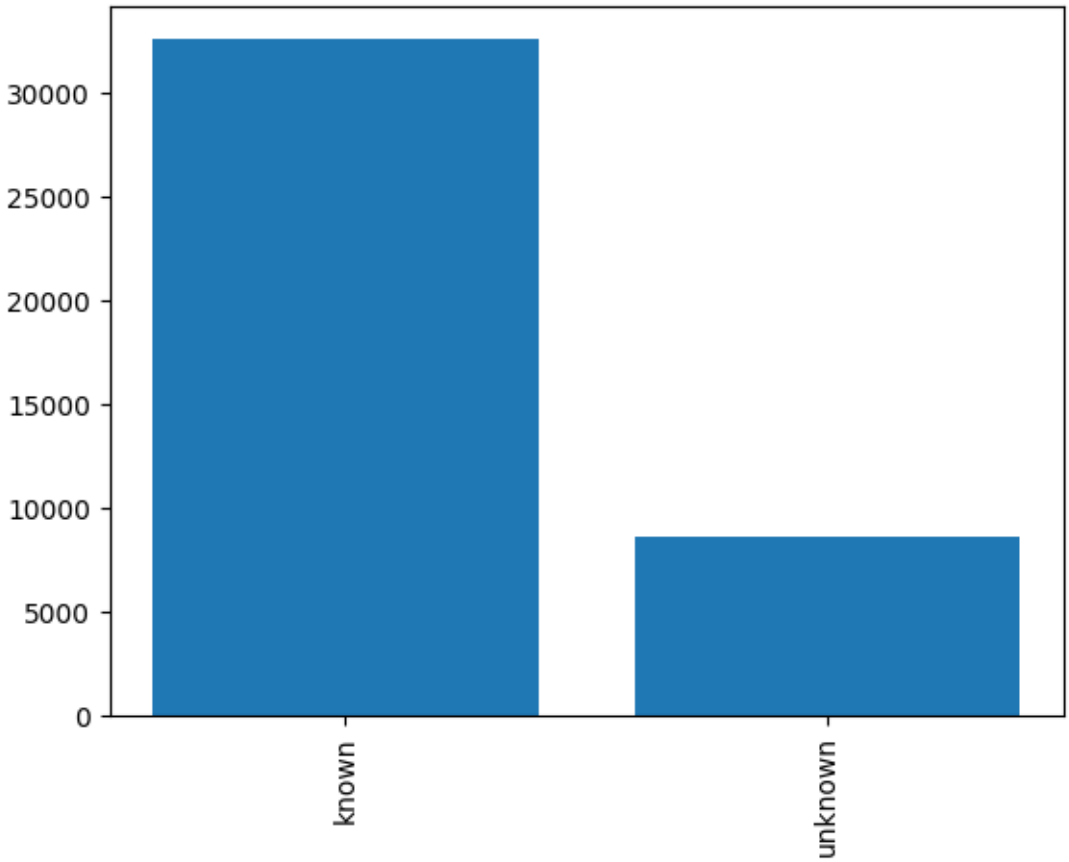
While this column sounds very helpful for predicting the target, the fact there are only three 'yes' values poses an issue when it comes to training the model. For one, there is a very high risk of overfitting using the 'yes' value, especially for tree models which could potentially create a split purely based on the 'yes' value and falsify its learning.

The solution to this is to consider the other valuable information that can be gathered from this

column, which is whether the value is known or unknown. While this may seem less directly relevant to predicting the 'subscribed' target column, it could actually be helpful. One example: If the bank has more information on the client, they could potentially pitch better to them during the campaign call and convince them easier. While this may not be true, this information is still worth letting the model learn from as it can simply choose to not use the information if it is truly irrelevant.

[37]:
```
# Combine 'yes' and 'no' into 'known'
df_credit.credit_default = df_credit.credit_default.apply(lambda x: x if x ==␣
 ↪"unknown" else "known")
print(df_credit.credit_default.value_counts())
plt.bar(df_credit.credit_default.value_counts().index, df_credit.credit_default.
 ↪value_counts().values)
plt.xticks(rotation=90)
plt.show()
```

```
credit_default
known      32591
unknown     8597
Name: count, dtype: int64
```



Summary In order to address the critically low number of 'yes' instances and prevent the model from overfitting on them, the 'yes' and 'no' categories are collapse into 'known' as analysis shows that this still gives very helpful information.

1.12 ### Housing Loan

Extremely large (>60%) of null values - Fill null values with 'missing'

---

[38]:
```
df_housing = df_credit.copy()
df_housing.head()
```

[38]:

	age	age_unk	occupation	marital_status	education_level	credit_default	\
0	57	False	technician	married	high.school	known	
1	55	False	unknown	married	unknown	unknown	
2	33	False	blue-collar	married	basic.9y	known	
3	36	False	admin.	married	high.school	known	
4	27	False	housemaid	married	high.school	known	

	housing_loan	personal_loan	contact_method	campaign_calls	\
0	no	yes	cellular	1	
1	yes	no	telephone	2	
2	no	no	cellular	1	
3	no	no	telephone	4	
4	None	no	cellular	2	

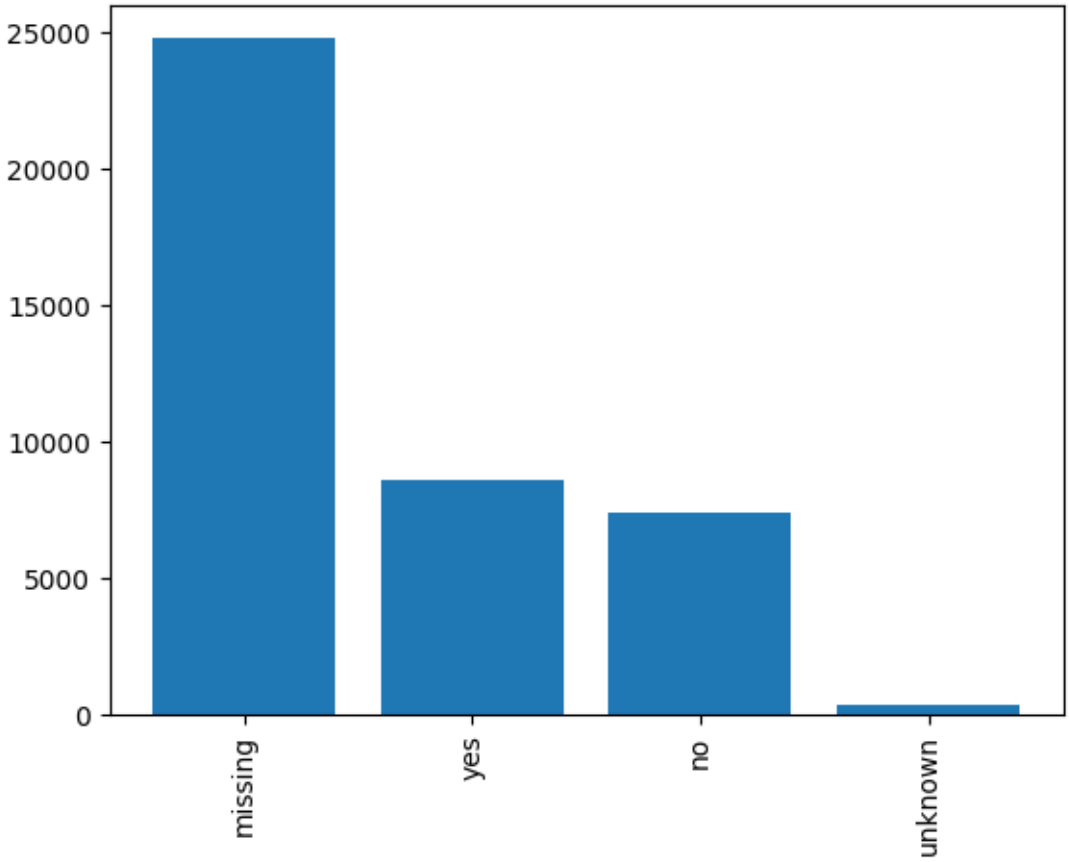
	previous_contact_days	previously_contacted	subscription_status
0	-1	False	no
1	-1	False	no
2	-1	False	no
3	-1	False	no
4	-1	False	no

[39]:
```
print(f"Number of null values in housing_loan column: {df_housing.housing_loan.
  ↪isna().sum()}")
print(df_housing.housing_loan.unique())
```

```
Number of null values in housing_loan column: 24789
['no' 'yes' None 'unknown']
```

[40]:
```
# Large amount of null values, fill with string 'mssing' instead (to allow␣
  ↪plotting without error)
df_housing.housing_loan = df_housing.housing_loan.fillna("missing")
print(df_housing.housing_loan.value_counts())
plt.bar(df_housing.housing_loan.value_counts().index, df_housing.housing_loan.
  ↪value_counts().values)
plt.xticks(rotation=90)
plt.show()
```

```
housing_loan
missing    24789
yes         8595
no          7411
unknown      393
Name: count, dtype: int64
```



Why no imputation/dropping? - Extremely high (>60%) of missing values is difficult to attempt to clean comprehensively - Dropping rows with null values is unviable due to the scale of them - Imputation is also very likely to introduce noise as there is not enough available data to learn patterns from - Leaving the 'missing' value inside the column is safer as the final model will more effectively learn what to do with it rather than risking introducing artificial patterns

Summary All missing values are imputed as 'missing' as many models cannot directly deal with null values

1.13 ### Personal Loan

Small number (~10%) of null values - mode imputation

---

[41]:
```
df_personal = df_housing.copy()
df_personal.head()
```

[41]:

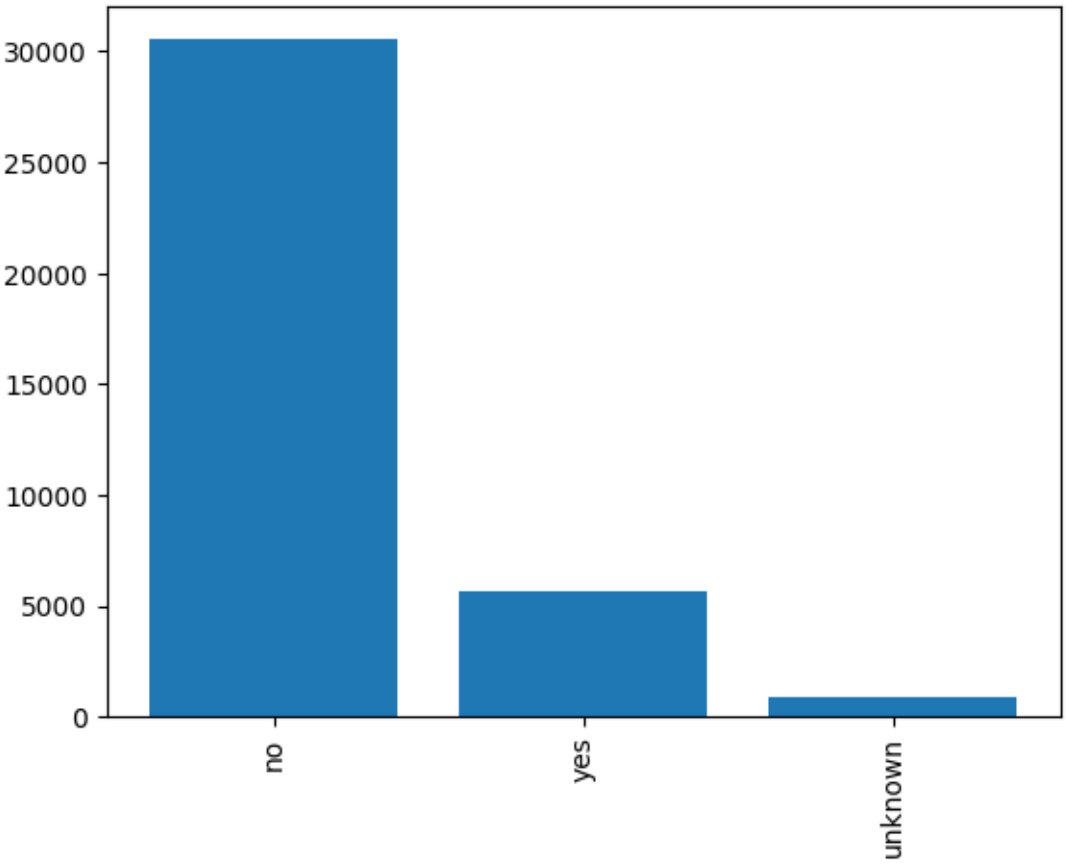
	age	age_unk	occupation	marital_status	education_level	credit_default	housing_loan	personal_loan	contact_method	campaign_calls	previous_contact_days	previously_contacted	subscription_status
0	57	False	technician	married	high.school	known	no	yes	cellular	1	-1	False	no
1	55	False	unknown	married	unknown	unknown	yes	no	telephone	2	-1	False	no
2	33	False	blue-collar	married	basic.9y	known	no	no	cellular	1	-1	False	no
3	36	False	admin.	married	high.school	known	no	no	telephone	4	-1	False	no
4	27	False	housemaid	married	high.school	known	missing	no	cellular	2	-1	False	no

[42]:
```
print(f"Number of null values in personal_loan column: {df_personal.
 ↪personal_loan.isna().sum()}")
# fillna for plotting, same as housing loan
df_personal.personal_loan = df_personal.personal_loan.fillna(df_personal.
 ↪personal_loan.mode())
print(df_personal.personal_loan.unique())
```

```
Number of null values in personal_loan column: 4146
['yes' 'no' nan 'unknown']
```

[43]:
```
print(df_personal.personal_loan.value_counts())
plt.bar(df_personal.personal_loan.value_counts().index, df_personal.
 ↪personal_loan.value_counts().values)
plt.xticks(rotation=90)
plt.show()
```

```
personal_loan
no         30532
yes         5633
unknown      877
Name: count, dtype: int64
```



Personal Loan has better options than just keeping the null values as missing, due to the fact that not that much data is missing. Dropping all the data will not be as destructive as the housing loan column, and imputation is likely safer as less of the data will be artificial and it won't destroy the real distribution if imputed values are inaccurate.

When it comes to imputation, mode imputation is the method that will be used. This is because 'no' is already te highest percentage value and there is little missing data, so filling all missing data with 'no' will also not disturb the distribution too much. It is also riskier to attempt to impute it using other columns like the age column because this column has less clear correlation with other columns.

Summary Used mode of column to impute missing values

1.14 ## Part 2c - Age Imputation (Darren's)

The goal is to determine whether age can be imputed based on data from other columns

---

[44]:
```
# Function to check one way ANOVA between integer column and categorical column
```

```
def get_one_way_anova(numerical_col: str, categorical_col: str, df: pd.
 ↪DataFrame):
    # Create list of numerical values for each category
    groups = [unique for unique in df[categorical_col].unique() if unique not␣
 ↪in ["unknown"]]
    group_data = [df[df[categorical_col] == g][numerical_col].dropna().
 ↪to_list() for g in groups]

    # Calculate one way anova
    f_stat, p_val = stats.f_oneway(*group_data, equal_var=False)
    return f_stat, p_val

# Function to get categorical distribution for a numerical column
def get_cat_distribution(numerical_col: str, categorical_col: str, df: pd.
 ↪DataFrame):
    # Create list of numerical values for each category
    groups = [unique for unique in df[categorical_col].unique() if unique not␣
 ↪in ["unknow", None]]
    group_data = [df[df[categorical_col] == g][numerical_col].dropna().
 ↪to_list() for g in groups]
    print(groups)

    # Plot histogram based on data points
    fig, axes = plt.subplots(1, len(group_data), figsize=(20, 5))
    for i, ax in enumerate(axes):
        ax.hist(group_data[i], bins=25, density=True)
        ax.set_title(groups[i])
    return fig

# Determine if normally distributed
def get_kwh_test(numerical_col: str, categorical_col: str, df: pd.DataFrame):
    # Create list of numerical values for each category
    groups = [unique for unique in df[categorical_col].unique() if unique not␣
 ↪in []]
    group_data = [df[df[categorical_col] == g][numerical_col].dropna().
 ↪to_list() for g in groups]

    # Calculate Kruskal-Wallis H test
    f_stat, p_val = stats.kruskal(*group_data)
    return f_stat, p_val
```

[45]:
```
df_age_impute = df_personal.copy()

# Modify dataframe to make it compatible with ANOVA
df_age_impute.occupation = df_age_impute.occupation.map(lambda cat: cat.lower().
 ↪replace(".", ""))
```

```
df_age_impute.education_level = df_age_impute.education_level.map(lambda cat:␣
 ↪cat.lower().replace(".", "_"))
df_age_impute.age = df_age_impute.age.map(lambda x: np.nan if x == -1 else x)
df_age_impute.head()
```

[45]:

	age	age_unk	occupation	marital_status	education_level	credit_default	housing_loan	personal_loan	contact_method	campaign_calls	previous_contact_days	previously_contacted	subscription_status
0	57.0	False	technician	married	high_school	known	no	yes	cellular	1	-1	False	no
1	55.0	False	unknown	married	unknown	unknown	yes	no	telephone	2	-1	False	no
2	33.0	False	blue-collar	married	basic_9y	known	no	no	cellular	1	-1	False	no
3	36.0	False	admin	married	high_school	known	no	no	telephone	4	-1	False	no
4	27.0	False	housemaid	married	high_school	known	missing	no	cellular	2	-1	False	no

[46]:
```
# # Plots all columns as histograms, omitted since output is very long.
# # Based on observations it is safe to say that

# for column in df_age_impute:
#     if column in ["age", "age_unk", "subscriber_status"]:
#         continue
#     get_cat_distribution("age", column, df_age_impute)
#     plt.show()
```

[47]:
```
df_age_impute.credit_default.unique()
```

[47]:
```
array(['known', 'unknown'], dtype=object)
```

[48]:
```
p_result = {"column": [], "p_value": [], "f_statistic": []}
# These ignore columns should have nothing to do with age at all
ignore_columns = ["age", "age_unk", "subscription_status", "campaign_calls",␣
 ↪"previous_contact_days", "previously_contacted"]
for column in df_age_impute:
    if column in ignore_columns:
        continue
```

```
    f_stat, p_val = get_kwh_test("age", str(column), df_age_impute)
    p_result["column"].append("age_" + str(column))
    p_result["p_value"].append(p_val)
    p_result["f_statistic"].append(f_stat)

p_results_df = pd.DataFrame(p_result)
p_results_df
```

```
/tmp/ipykernel_101998/1416991104.py:32: SmallSampleWarning: One or more sample
arguments is too small; all returned values will be NaN. See documentation for
sample size requirements.
  f_stat, p_val = stats.kruskal(*group_data)
```

[48]:

	column	p_value	f_statistic
0	age_occupation	0.000000e+00	6194.225900
1	age_marital_status	0.000000e+00	7500.062350
2	age_education_level	0.000000e+00	2246.020546
3	age_credit_default	2.791578e-295	1348.811949
4	age_housing_loan	6.821295e-01	1.500610
5	age_personal_loan	NaN	NaN
6	age_contact_method	8.883125e-11	42.053034

Explanation & Thought Process:

With many missing values in the age column, the goal here is to impute the missing values using the data contained in the other columns. The reason why I think this may be possible for the age column is due to the fact that it may be possible to impute age based on data such as their occupation or marital status.

The first step to decide which columns' data I would use for the imputation is through the one-way ANOVA method, which tests whether the means of several independent categories have statistically significant differences. This can help to determine whether the specified categories have an influence on a dependent numerical (integer) value, in this case being the age column.

I did this by splitting a data column, for example: marital_status into its unique categories (married, single, divorced), before finding all the ages associated with this category. From there, I can caluclate the one way anova using scipy.stats.f_oneway(). However, upon observation of the histograms of each category inside each of the data columns, I realised that the data does not fulfil some of the requirements of using the one-way ANOVA test. The first requirement being that the dependent variable (age) should be normally distributed across all sample groups (categories). The second being that the variances should be roughly equal across all groups.

Hence, I decided to use a non-parametric version of one-way anova, called the Kruskal-Wallis Test, which instead determines if the medians of several independent categories have statistically significant differences. I implemented this test using scipy.stats.kruskal()

Application:

I looped through the likely candidate columns that may affect the age column, which were everything except age, subscriber_status. campaign_calls and previous_contact_days were not test because they were numerical columns and hence would break one requirement of the test, which

was that each group needed at least 5 observations. This was in conjunction with the fact that these data points should not affect the age except through dubious / coincidental statistical correlation since this data is taken by the bank itself.

For the Kruskal-Wallis Test, we define 2 hypotheses: Null Hypothesis: The population distributions / medians are all equal (no differences / no influence) Alternative Hypothesis: At least one of the population medians / distributions are different from the others (has influence)

If the p-value <= alpha (0.05) we can safely reject the null hypothesis. If the p-value > alpha, we will fail to reject the null hypothesis.

As seen in the above p-values, there are 5 columns that have the potential to help with the imputation of unknown values in the age column. These 5 columns being occupation, marital_status, education_level, credit_defaut, and contact_method. This is because their p-value is less than our alpha (set at 0.05) and thus we can reject the null hypothesis for these columns. This means that these 5 columns are likely to have an influence on the age specified in the age column. Thus, I will use these columns for imputation via KNN.

1.14.1 Imputation

[49]:
```
df_age_impute_mean = df_personal.copy()
print(df_age_impute_mean.loc[df_age_impute_mean.age == -1].shape[0])

# NOTE: Including unknown values in the data does not negatively affect final␣
 ↪distribution
df_age_impute_mean = df_age_impute_mean.iloc[:, [0, 2, 3, 4, 5, 8]]
print(df_age_impute_mean.loc[df_age_impute_mean.age == -1].shape[0])
```

```
4197
4197
```

[50]:
```
# Imputing the data
df_age_impute_ohe = pd.get_dummies(df_age_impute_mean, ["occupation",␣
 ↪"marital_status", "education_level", "credit_default", "contact_method"])
df_age_impute_ohe.age = df_age_impute_ohe.age.map(lambda x: x if x != -1 else␣
 ↪np.nan)
df_age_impute_ohe.age.isnull().sum()

imputer = KNNImputer(n_neighbors=2, weights="distance")
imputed_data = imputer.fit_transform(df_age_impute_ohe.reset_index().
 ↪drop("index", axis=1))

imputed_df = pd.DataFrame(imputed_data, columns=df_age_impute_ohe.columns,␣
 ↪index=df_age_impute_ohe.index)
imputed_df.head()
```

[50]:

	age	occupation_admin.	occupation_blue-collar	occupation_entrepreneur	\
0	57.0	0.0	0.0	0.0	
1	55.0	0.0	0.0	0.0	

```
2  33.0                0.0                      1.0                      0.0
3  36.0                1.0                      0.0                      0.0
4  27.0                0.0                      0.0                      0.0

   occupation_housemaid  occupation_management  occupation_retired  \
0                   0.0                    0.0                 0.0
1                   0.0                    0.0                 0.0
2                   0.0                    0.0                 0.0
3                   0.0                    0.0                 0.0
4                   1.0                    0.0                 0.0

   occupation_self-employed  occupation_services  occupation_student  …  \
0                       0.0                  0.0                 0.0  …
1                       0.0                  0.0                 0.0  …
2                       0.0                  0.0                 0.0  …
3                       0.0                  0.0                 0.0  …
4                       0.0                  0.0                 0.0  …

   education_level_basic.9y  education_level_high.school  \
0                       0.0                          1.0
1                       0.0                          0.0
2                       1.0                          0.0
3                       0.0                          1.0
4                       0.0                          1.0

   education_level_illiterate  education_level_professional.course  \
0                         0.0                                  0.0
1                         0.0                                  0.0
2                         0.0                                  0.0
3                         0.0                                  0.0
4                         0.0                                  0.0

   education_level_university.degree  education_level_unknown  \
0                                0.0                      0.0
1                                0.0                      1.0
2                                0.0                      0.0
3                                0.0                      0.0
4                                0.0                      0.0

   credit_default_known  credit_default_unknown  contact_method_cellular  \
0                   1.0                    0.0                      1.0
1                   0.0                    1.0                      0.0
2                   1.0                    0.0                      1.0
3                   1.0                    0.0                      0.0
4                   1.0                    0.0                      1.0

   contact_method_telephone
```

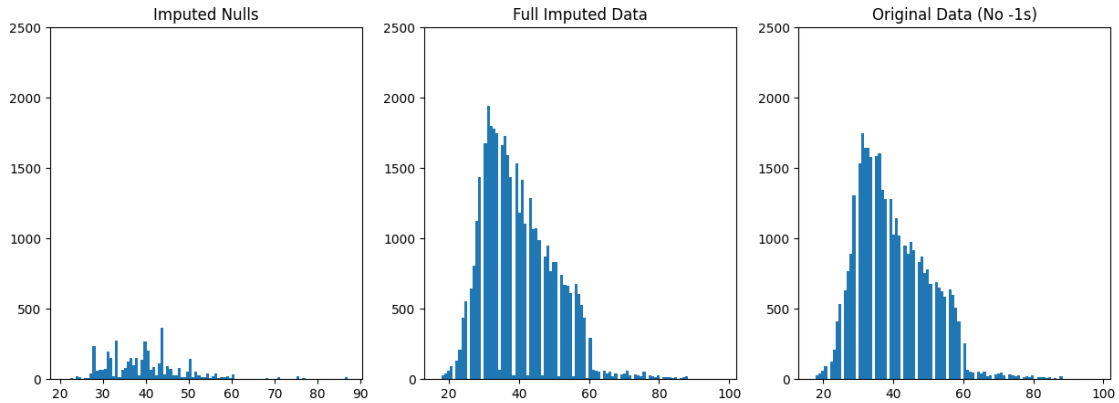
```
0                       0.0
1                       1.0
2                       0.0
3                       1.0
4                       0.0

[5 rows x 29 columns]
```

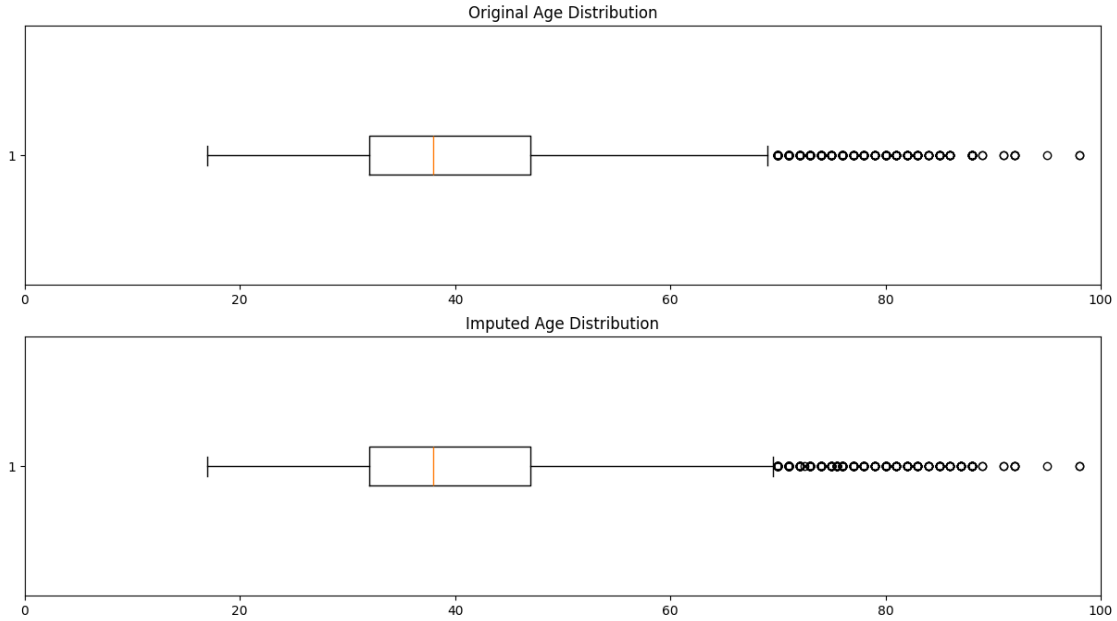
[51]:
```
# Based on graphs and describe() below, the overall distribution, IQR, mean,␣
 ↪std are very similar to the original datasets (respectively).
# With this it is safe to assume that the imputation of the age column was␣
 ↪successful
# It was not possible to test distribution similarity since the KS test␣
 ↪requires continuous distributions (original is not coninuous)
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
ax[0].hist(imputed_df.loc[df_age_impute_ohe.loc[df_age_impute_ohe.age.isnull()].
 ↪index].age, bins=100)
ax[1].hist(imputed_df.age, bins=100)
ax[2].hist(df_age_impute_ohe.age, bins=100)
ax[0].set_title("Imputed Nulls")
ax[1].set_title("Full Imputed Data")
ax[2].set_title("Original Data (No -1s)")
for x in ax:
    x.set_ylim(0, 2500)
plt.plot()

fig, ax = plt.subplots(2, 1, figsize=(15, 8))
ax[0].boxplot(df_age_impute_ohe.loc[df_age_impute_ohe.age.isna() == False].age,␣
 ↪vert=False)
ax[1].boxplot(imputed_df.age, vert=False)
ax[0].set_title("Original Age Distribution")
ax[1].set_title("Imputed Age Distribution")
for x in ax:
    x.set_xlim(0, 100)
plt.plot()

print("Original Data (No -1s)")
print(df_age_impute_ohe.loc[df_age_impute_ohe.age.isna() == False].age.
 ↪describe())
print("\nImputed Data")
print(imputed_df.age.describe())

df_final_age_imputed = imputed_df.age.astype(int)
# Change age to reflect new imputed data
df_final_age_imputed.head()
```

```
Original Data (No -1s)
```

```
count    36991.000000
mean        40.025303
std         10.436777
min         17.000000
25%         32.000000
50%         38.000000
75%         47.000000
max         98.000000
Name: age, dtype: float64

Imputed Data
count    41188.000000
mean        40.001505
std         10.311787
min         17.000000
25%         32.000000
50%         38.000000
75%         47.000000
max         98.000000
Name: age, dtype: float64
```

[51]:
```
0    57
1    55
2    33
3    36
4    27
Name: age, dtype: int64
```





1.15 ## Part 3 - Bivariate Analysis (Harish's)

Purpose - Check out how each feature could help in predicting the target column - Identify targets for feature selection/engineering if needed

---

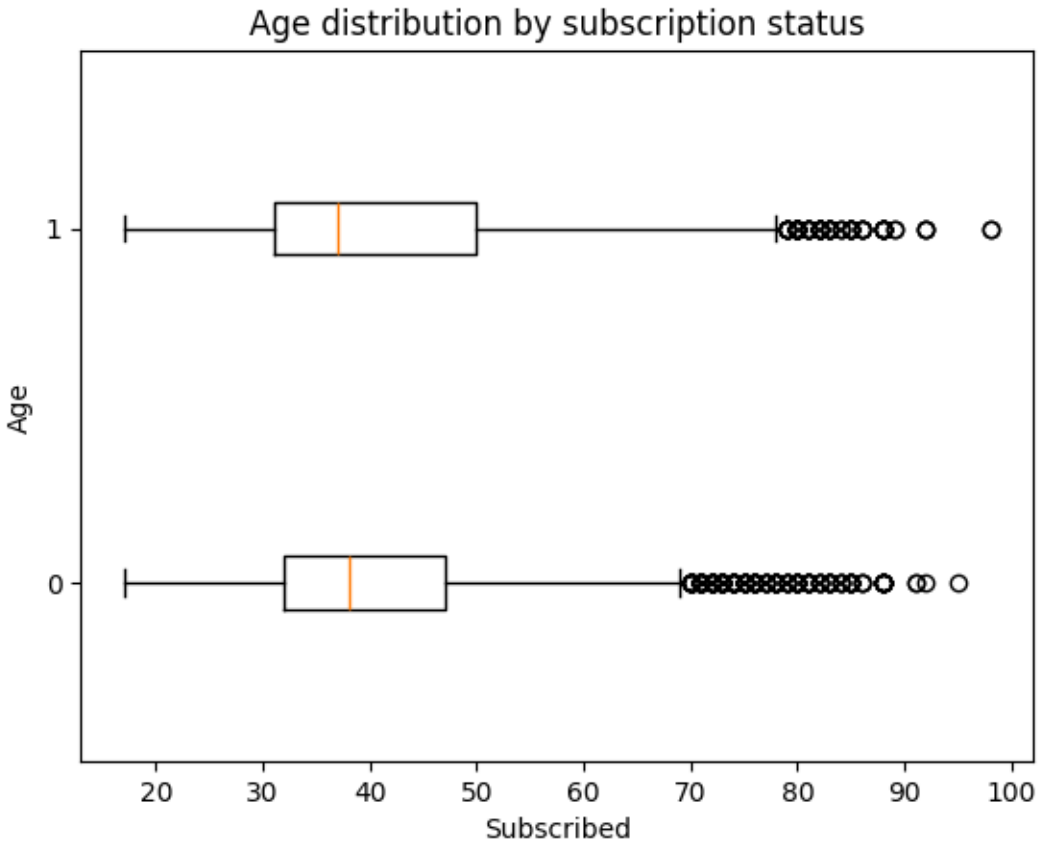
[52]:
```
df_cleaned = df_personal.copy()
df_cleaned.subscription_status = df_cleaned.subscription_status.apply(lambda x:␣
 ↪True if x == "yes" else False)
df_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   age                   41188 non-null  int64
 1   age_unk               41188 non-null  bool
 2   occupation            41188 non-null  object
 3   marital_status        41188 non-null  object
 4   education_level       41188 non-null  object
 5   credit_default        41188 non-null  object
 6   housing_loan          41188 non-null  object
 7   personal_loan         37042 non-null  object
 8   contact_method        41188 non-null  object
 9   campaign_calls        41188 non-null  int64
 10  previous_contact_days 41188 non-null  int64
```

```
 11  previously_contacted   41188 non-null  bool
 12  subscription_status    41188 non-null  bool
dtypes: bool(3), int64(3), object(7)
memory usage: 3.3+ MB
```

1.15.1 Age

[53]:
```
# Eyeballing it the only difference seems to be visible at older ages that are
# already outliers as known from the univariate analysis
df_temp_age_cleaned = df_cleaned[df_cleaned['age'] != -1]

plt.figure()
plt.boxplot(
    [df_temp_age_cleaned[df_temp_age_cleaned.subscription_status == v].age
     for v in [0, 1]],
    tick_labels=["0", "1"],
    vert=False
)
plt.ylabel("Age")
plt.xlabel("Subscribed")
plt.title("Age distribution by subscription status")
plt.show()
```



[54]:
```
# There seems to be only a very small increase in the mean age for those who␣
 ↪subscribed
# There is no direct correlation between age and subscription status
# While the model can potentially gain insights from it, it is hard to tell␣
 ↪whether it'll be helpful
print("Subscribed = True")
df_temp_age_cleaned = df_cleaned[df_cleaned['age'] != -1]
print(df_temp_age_cleaned[df_temp_age_cleaned.subscription_status == True].age.
 ↪describe())
print("\nSubscribed = False")
print(df_temp_age_cleaned[df_temp_age_cleaned.subscription_status == False].age.
 ↪describe())
```

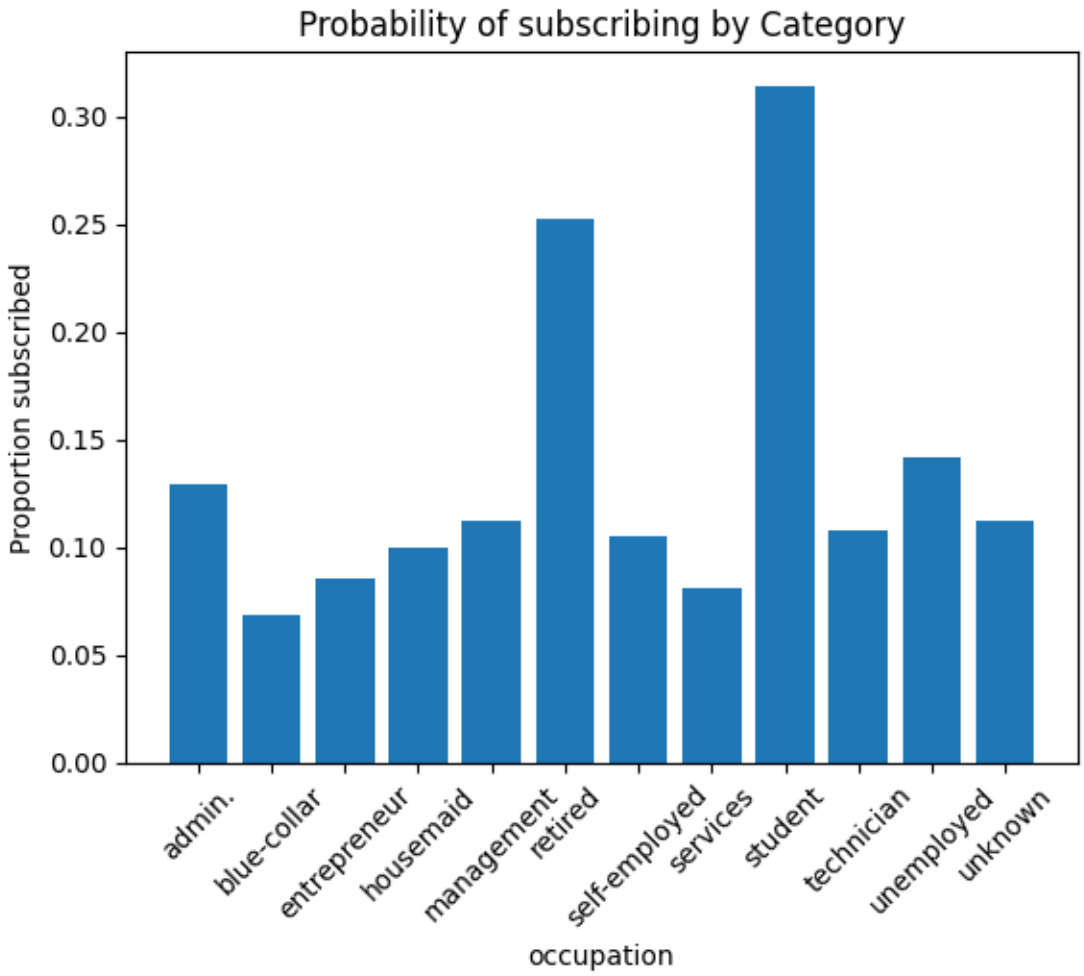
```
Subscribed = True
count    4191.00000
mean       40.93486
std        13.90226
min        17.00000
25%        31.00000
50%        37.00000
75%        50.00000
max        98.00000
Name: age, dtype: float64

Subscribed = False
count    32800.000000
mean        39.909085
std          9.901279
min         17.000000
25%         32.000000
50%         38.000000
75%         47.000000
max         95.000000
Name: age, dtype: float64
```

1.15.2 Occupation

[55]:
```
# Making a generic helper function to help with bivariate analysis
def bivariate_categorical_plotter(df, colname):
    prop = df.groupby(colname)["subscription_status"].mean()

    plt.figure()
    plt.bar(prop.index, prop.values)
    plt.ylabel("Proportion subscribed")
    plt.xlabel(colname)
```

```
    plt.xticks(rotation=45)
    plt.title("Probability of subscribing by Category")
    plt.show()
```

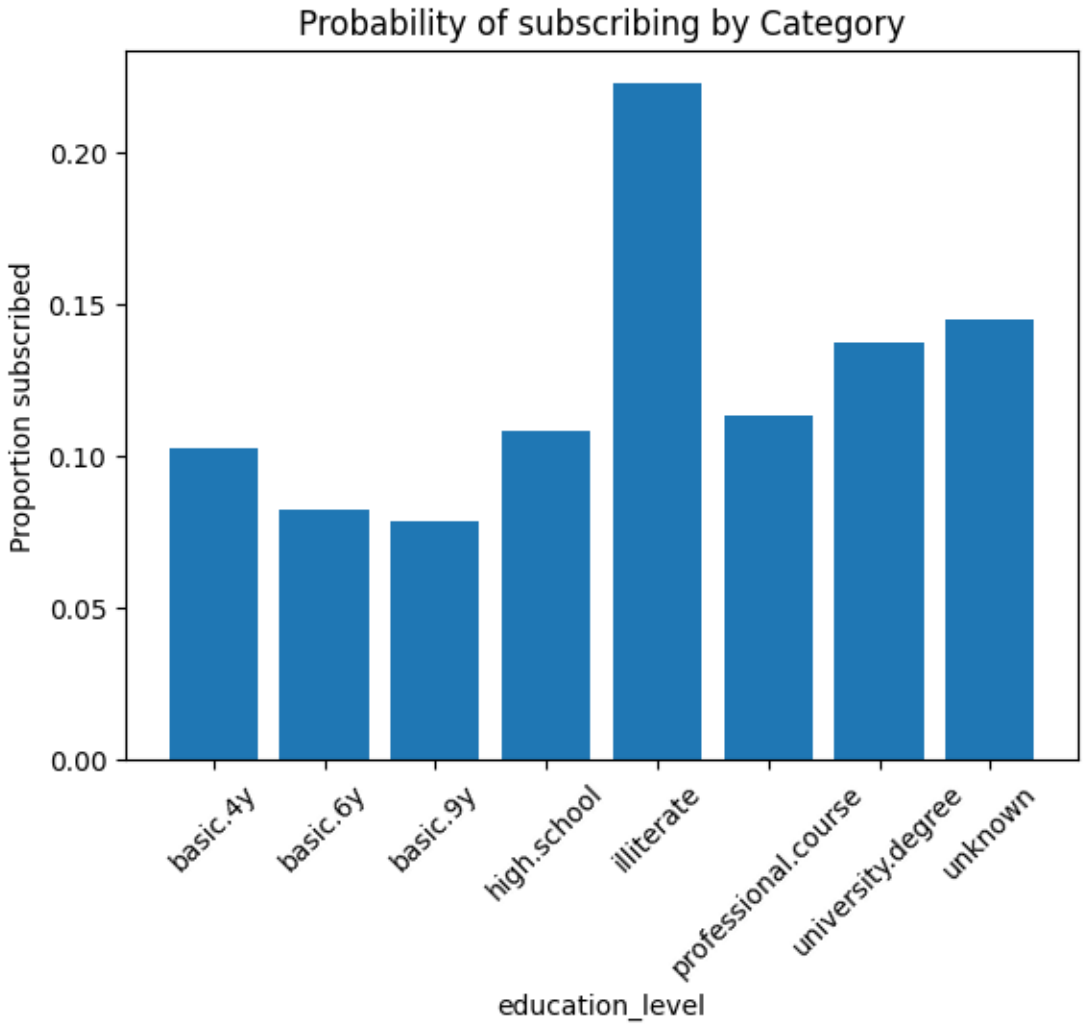
[56]:
```
# Looks to be a very good feature, as there are clear differences in the␣
 ↪proportion of
# people who subscribe based on their occupation.
bivariate_categorical_plotter(df_cleaned, "occupation")
```



1.15.3 Education Level

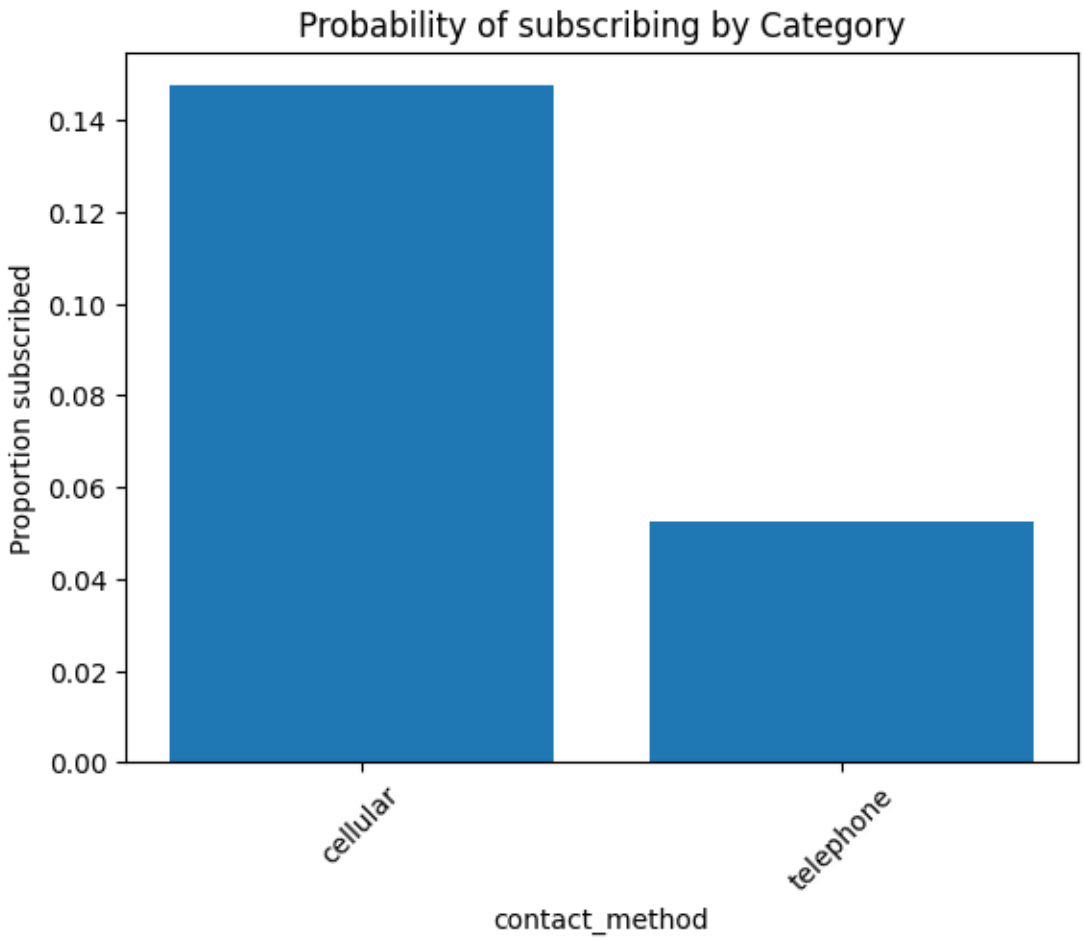
[57]:
```
# This feature also shows potential
# Small caveat is that the 'illiterate' value was a rare value as known from␣
 ↪univariate analysis,
# so that obvious probability jump can't be taken advantage of too much by the␣
 ↪model
```

```
bivariate_categorical_plotter(df_cleaned, "education_level")
```



1.15.4 Contact Method

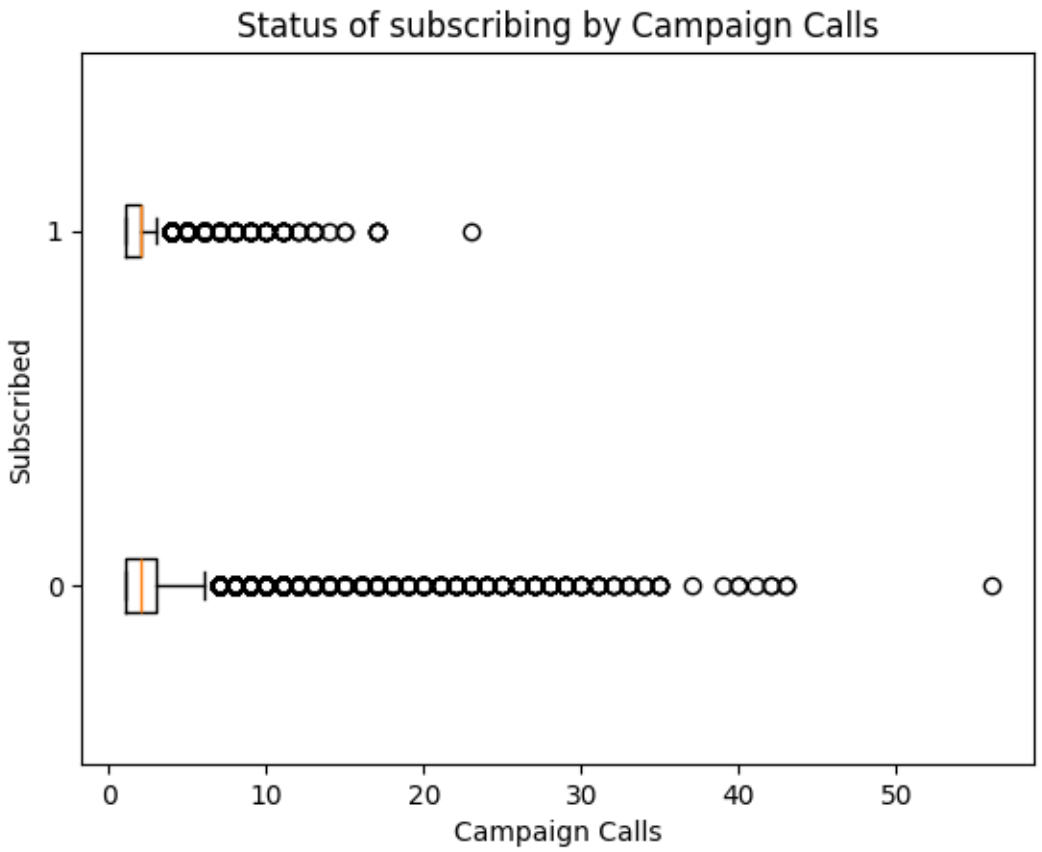
[58]:
```
# Quite a large difference between the probability of subscribing for both␣
↪contact methods
# Furthermore, both contact methods each share a large proportion of the data
# That makes this feature very helpful due to the large split
bivariate_categorical_plotter(df_cleaned, "contact_method")
```



1.15.5 Campaign Calls

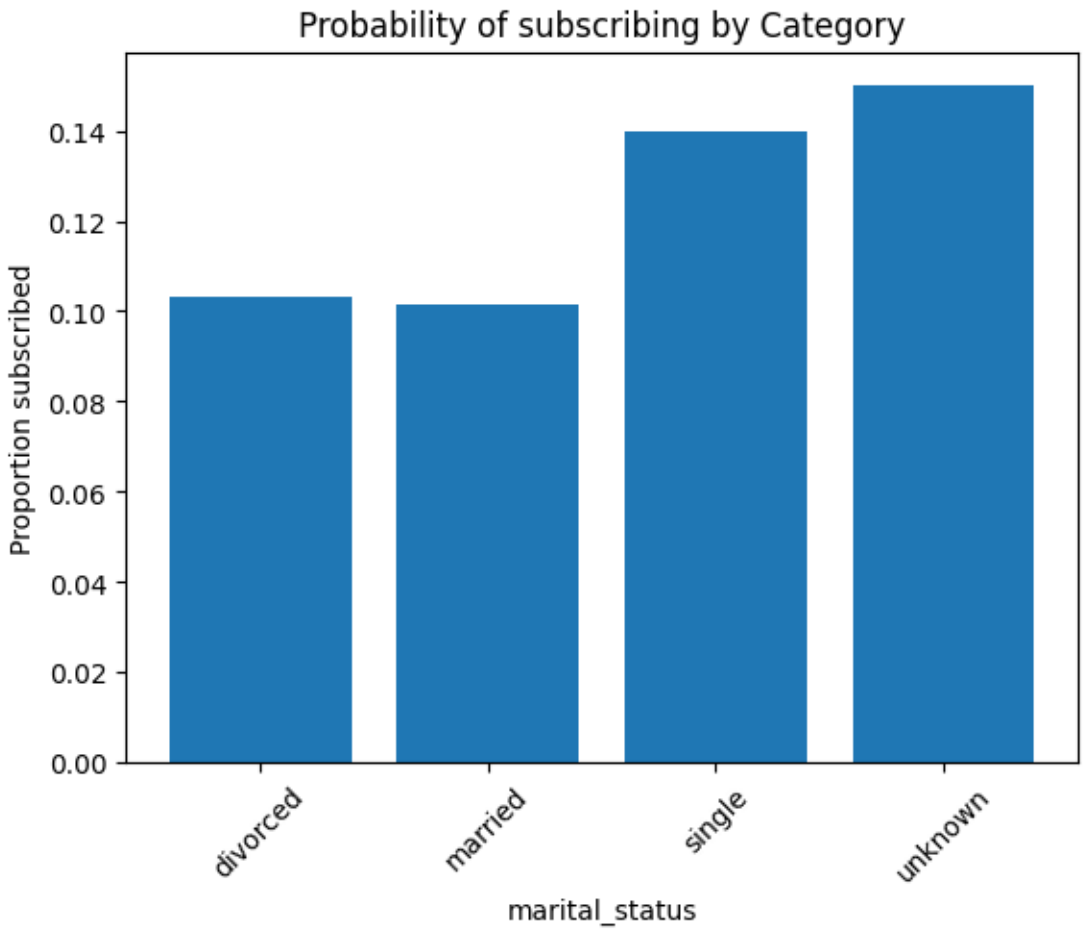
[59]:
```
# This feature is harder to make sense of, but my first guess is that after a␣
↪certain
# amount of calls, calling them more is very unlikely to change their mind
plt.boxplot(
    [df_cleaned[df_cleaned.subscription_status == v].campaign_calls
     for v in [0, 1]],
    tick_labels=["0", "1"],
    vert=False
)
plt.ylabel("Subscribed")
plt.xlabel("Campaign Calls")
plt.title("Status of subscribing by Campaign Calls")
```

[59]:
```
Text(0.5, 1.0, 'Status of subscribing by Campaign Calls')
```



1.15.6 Marital Status

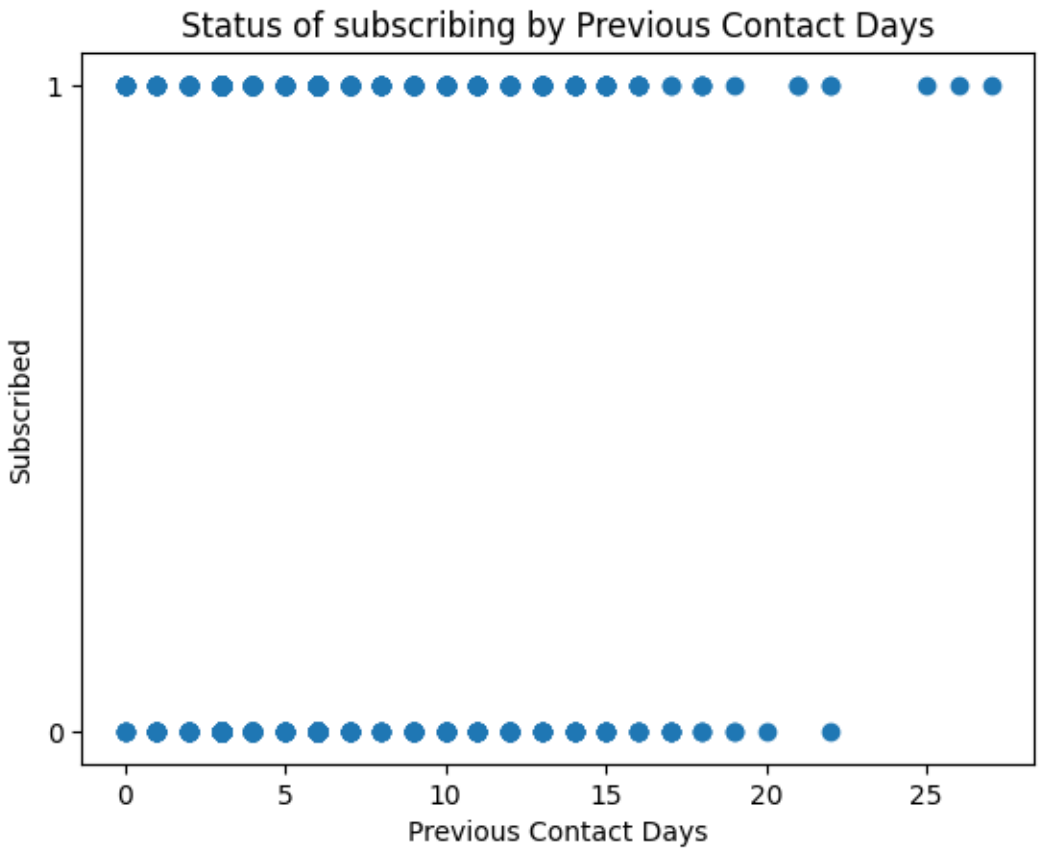
[60]:
```
# This one is weird because it is hard to interpret why 'unknown' could␣
 ↪possibly have
# the highest proportion of subscribing. But there were very few unknown values␣
 ↪so this
# could just be a less stable column. Still a helpful feature as it shows␣
 ↪single people
# are more likely to subscribe over people who are or were married.
bivariate_categorical_plotter(df_cleaned, "marital_status")
```



1.15.7 Previous Contact Days

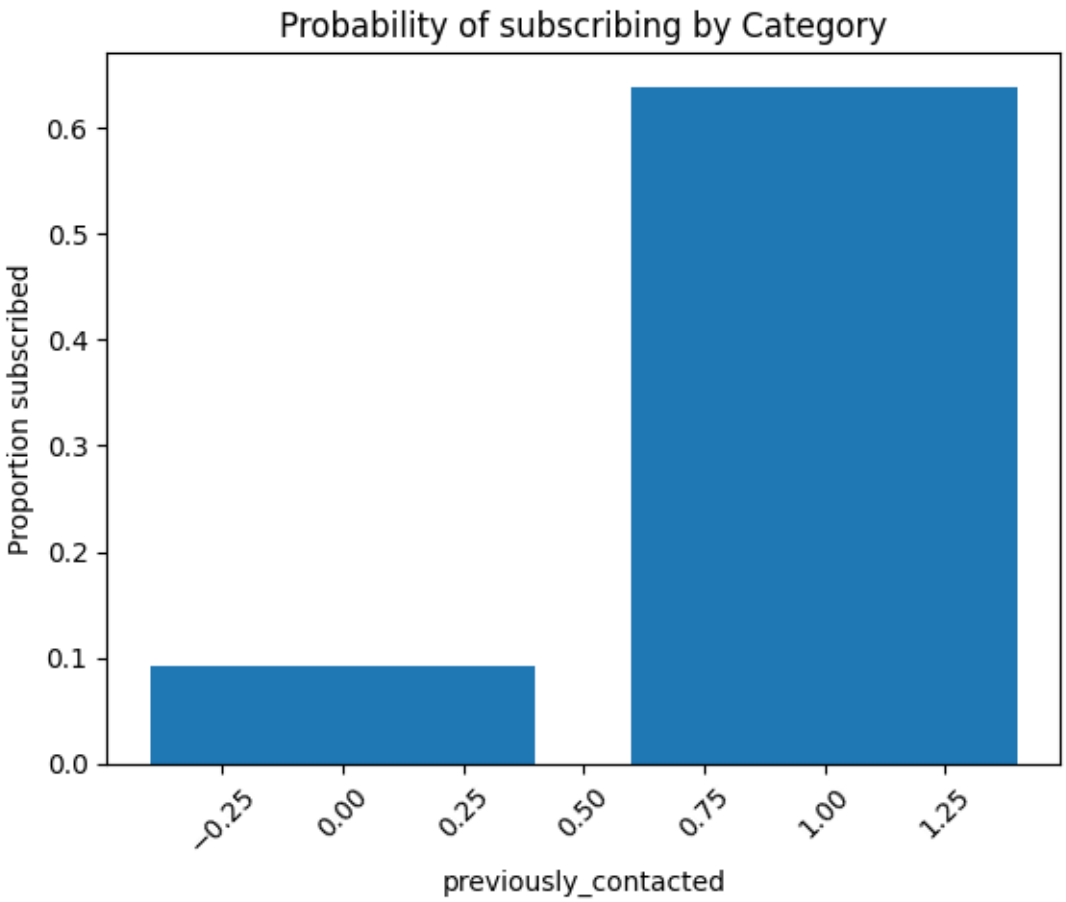
[ ]:
```
# Doesn't look like a helpful feature on its own
# It gives different information from previously_contacted
# for people who have been contacted before, but it seems useless
# and just noisy looking at the graph below
df_temp_pdays_cleaned = df_cleaned[df_cleaned['previous_contact_days'] != -1]

plt.figure()
plt.scatter(df_temp_pdays_cleaned.previous_contact_days, df_temp_pdays_cleaned.
  ↪subscription_status)
plt.yticks([0, 1])
plt.ylabel("Subscribed")
plt.xlabel("Previous Contact Days")
plt.title("Status of subscribing by Previous Contact Days")
```

[ ]:
```
Text(0.5, 1.0, 'Status of subscribing by Previous Contact Days')
```



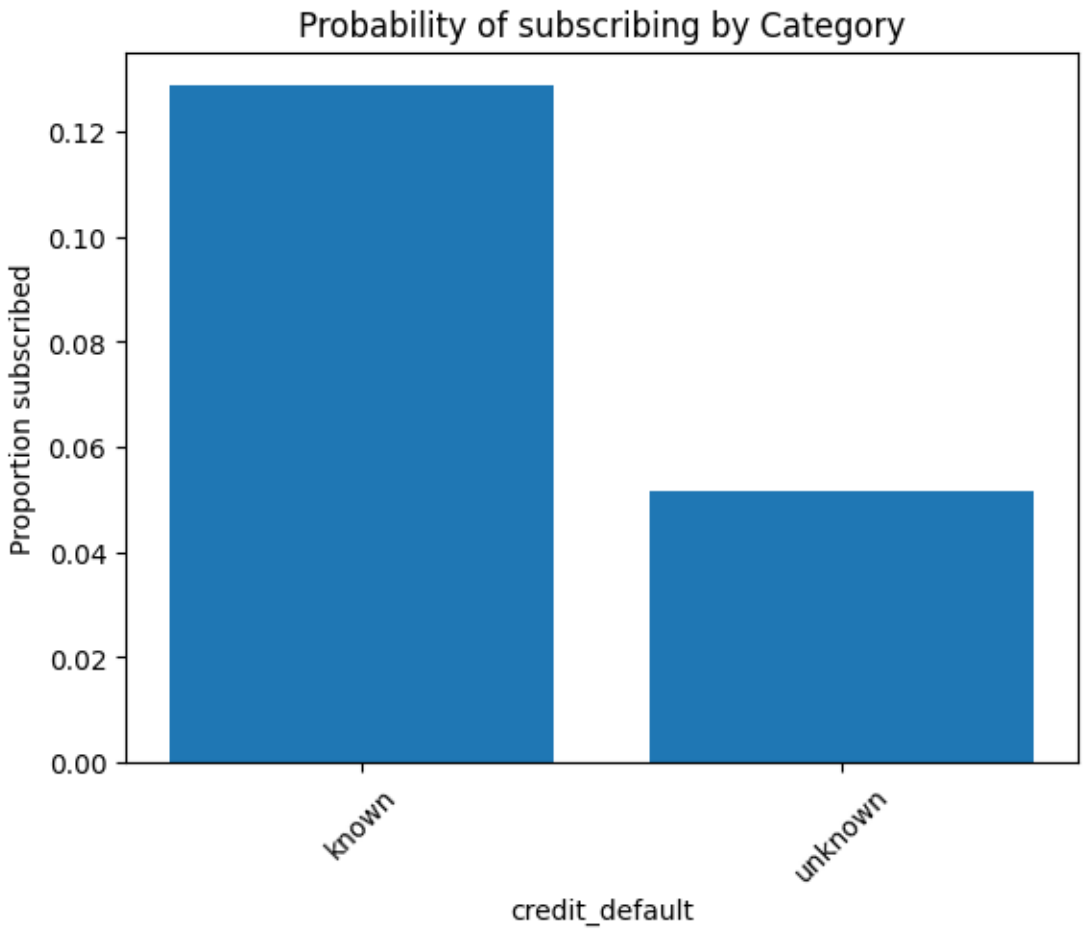
1.15.8 Previously Contacted

[ ]:
```
# Extremely helpful feature, with the most massive probability difference
# only disadvantage is that only about 5% have been previously contacted,
# but will be very helpful at identifying those people as likely to subscribe␣
 ↪for the model
bivariate_categorical_plotter(df_cleaned, "previously_contacted")
```



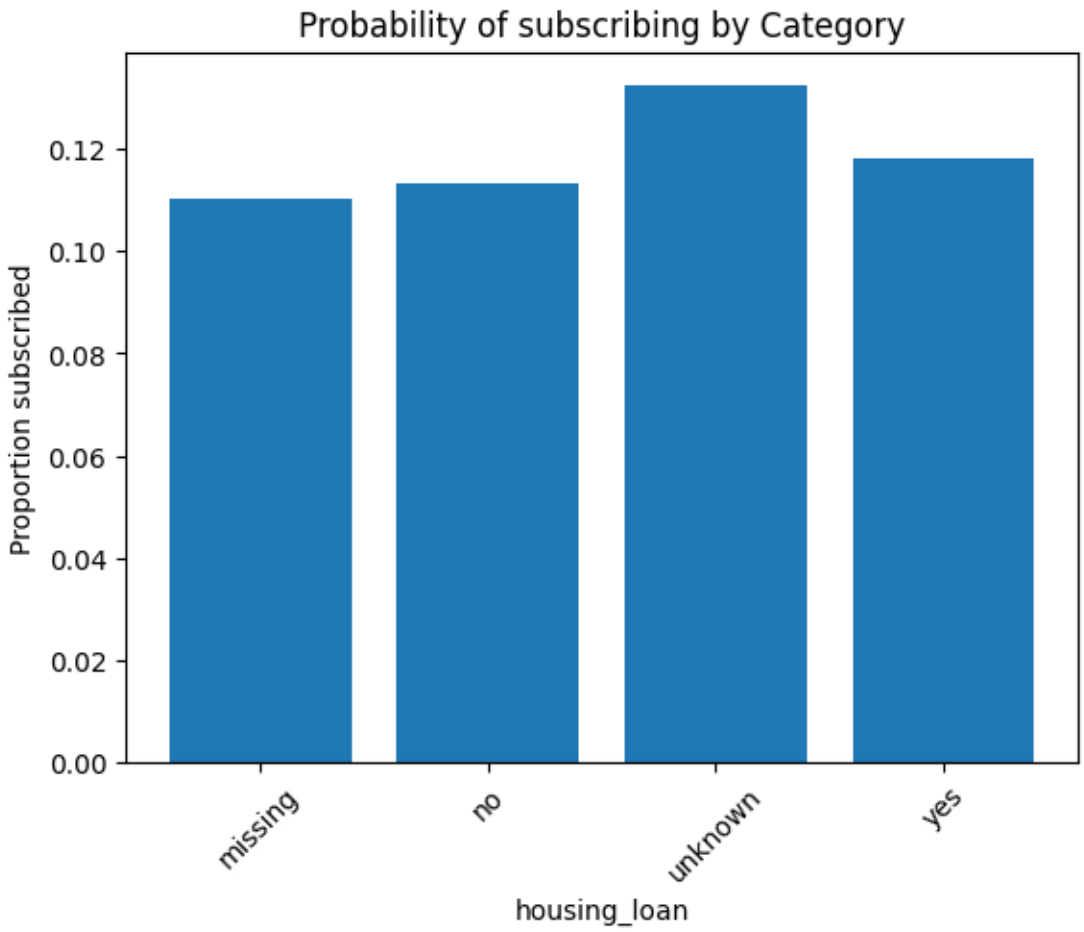
1.15.9 Credit Default

[62]:
```
# Overall seems to be a helpful feature, not much to say about it
# Obvious jump in subscription probability for those who have a known status
bivariate_categorical_plotter(df_cleaned, "credit_default")
```



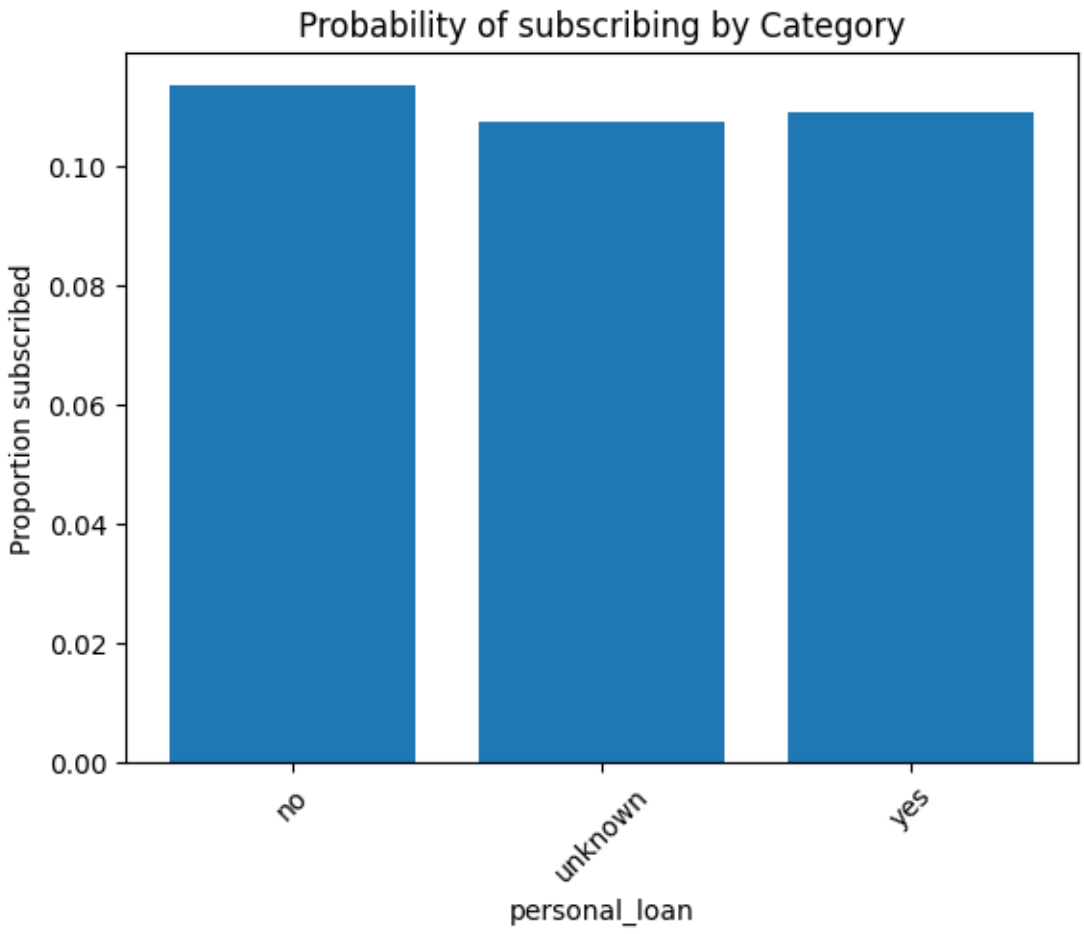
1.15.10 Housing Loan

[63]:
```
# Very small differences in probability
# Similar to Marital Status, the small amount of unknown values might have an␣
 ↪unstable value
# and hence is hard to interpret
# The values that are easy to assume that are the most helpful, which are yes␣
 ↪and no,
# seem to have the smallest difference in probability
# Not sure if this entire feature is worth keeping, when we already know 60% of␣
 ↪the data is missing
# and the non-missing data seems to not help too much
bivariate_categorical_plotter(df_cleaned, "housing_loan")
```



1.15.11 Personal Loan

[64]:
```
# Similar to housing loan, there are very small differences in probability
# This column might also be worth removing as a feature as it could potentially␣
  ↪just be noise
bivariate_categorical_plotter(df_cleaned, "personal_loan")
```



1.16 ## Part 4 - Feature Engineering

Purpose - Rank features and select ones that are more likely to be use for the model - Create new features from existing features if possible to create new, relevant information — NOTE: Some of the actions taken in Data Cleaning could be considered as feature engineering, but this section is separate as it is intentionally trying to glean extra information from already clean columns, while feature engineering in the Data Cleaning section was compulsory in order to clean the data.

1.16.1 The Loan Columns

The two loan columns are the first target for this section as at first glance, these two columns did not seem like they would be individually strong for predicting the target.

Since both loan columns are primarily boolean values, one possible feature to engineer would be whether the client is confirmed to have at least one loan.

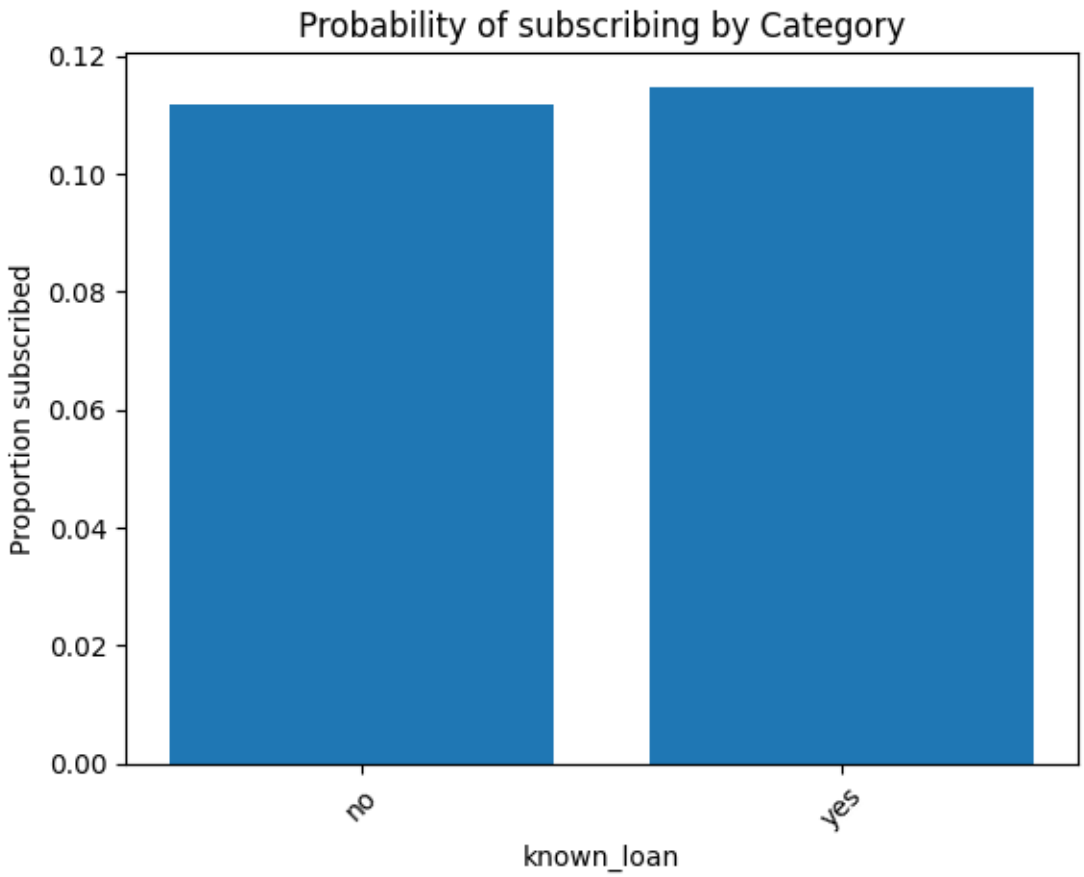
[65]:
```
df_features = df_cleaned.copy()
df_features.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 13 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   age                    41188 non-null  int64
 1   age_unk                41188 non-null  bool
 2   occupation             41188 non-null  object
 3   marital_status         41188 non-null  object
 4   education_level        41188 non-null  object
 5   credit_default         41188 non-null  object
 6   housing_loan           41188 non-null  object
 7   personal_loan          37042 non-null  object
 8   contact_method         41188 non-null  object
 9   campaign_calls         41188 non-null  int64
 10  previous_contact_days  41188 non-null  int64
 11  previously_contacted   41188 non-null  bool
 12  subscription_status    41188 non-null  bool
dtypes: bool(3), int64(3), object(7)
memory usage: 3.3+ MB
```

[66]:
```
df_features['known_loan'] = df_features.apply(lambda x: "yes" if (x.
 ↪housing_loan == "yes") \
                                              or (x.personal_loan == "yes") \
                                               else "no", axis=1)

df_features.known_loan.describe()
```

[66]:
```
count     41188
unique        2
top          no
freq      28292
Name: known_loan, dtype: object
```

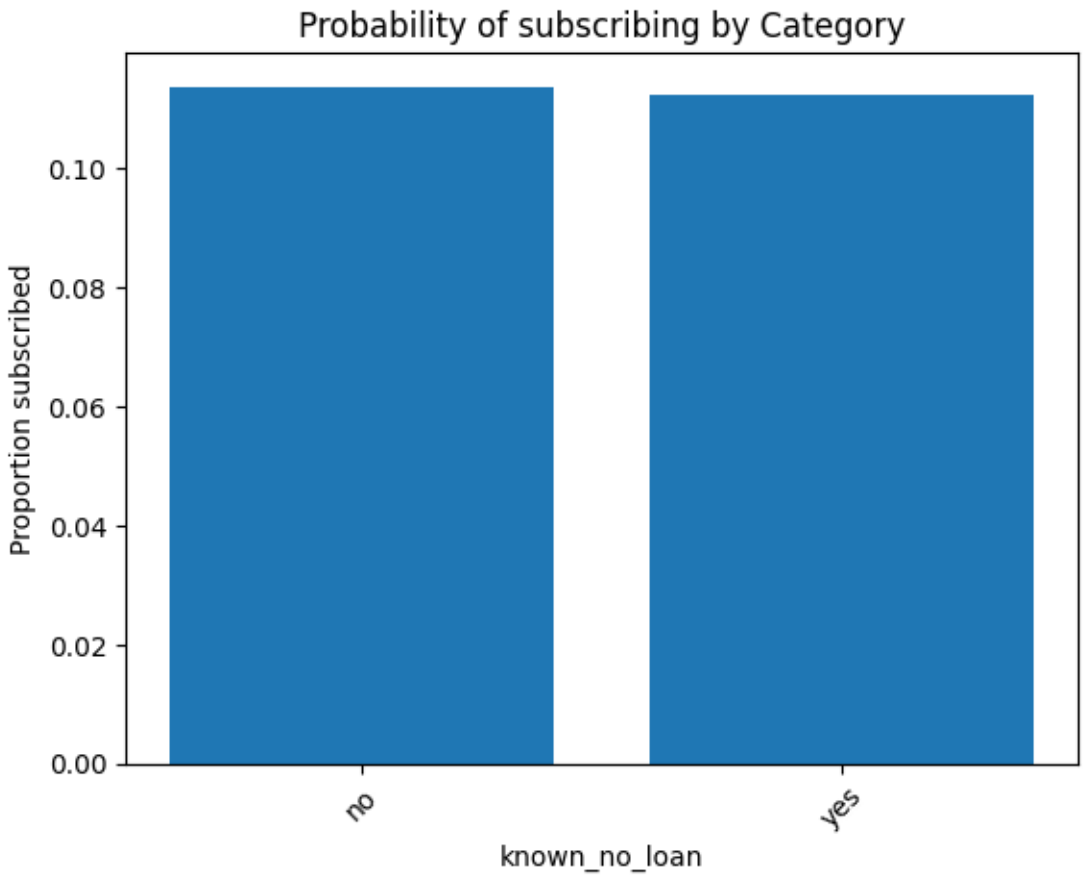
[67]:
```
bivariate_categorical_plotter(df_features, "known_loan")
```



[ ]:
```
# trying it the other way around
df_features['known_no_loan'] = df_features.apply(lambda x: "no" if (x.
 ↪housing_loan == "no") \
                                                 and (x.personal_loan == "no") \
                                                  else "yes", axis=1)
df_features.known_no_loan.describe()
```

[ ]:
```
count     41188
unique        2
top         yes
freq      35471
Name: known_no_loan, dtype: object
```

[69]:
```
bivariate_categorical_plotter(df_features, "known_no_loan")
```



Summary Feature engineering with the loan columns is unsuccessful and will not be added as new features to the model. The variation in distributions is too little and is more likely to be noise that the model will be forced to compute.

1.16.2 Occupation

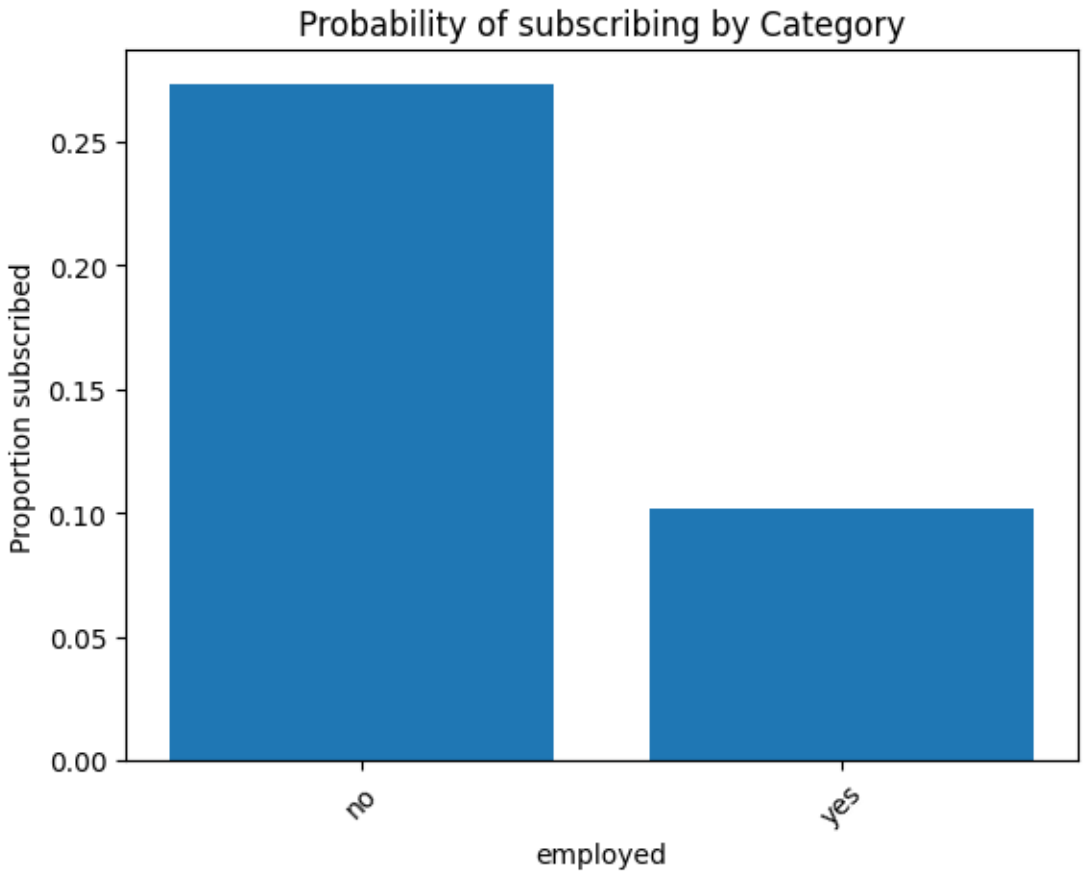
From bivariate analysis, it was noticed that rows with 'student' and 'retired' in the occupation column had the highest probability of subscribing with quite a big jump over all the other values which were normal jobs. This could potentially be leveraged to make a new, simple column that the model can use easily.

[70]:
```
df_features['employed'] = df_features.apply(lambda x: "no" if (x.occupation ==␣
↪"student") \
                                            or (x.occupation == "retired") \
                                             else "yes", axis=1)
df_features.employed.describe()
```

[70]:
```
count     41188
unique        2
```

```
top         yes
freq      38593
Name: employed, dtype: object
```

[71]:
```
bivariate_categorical_plotter(df_features, "employed")
```



Summary This engineered feature seems helpful for replacing the occupation feature. It retains the big distribution split between its classes, while significantly reducing the number of classes which should help the model discover splits faster and positively impact performance. The Kedro pipeline will replace the 'occupation' column with this new 'employed' column.

1.16.3 Feature Selection

The goal of this section is to create and justify an arbitrary ranking of features, and create a list ordered by rank.

This feature selection is more biased as only the direct correlation between the features and the target was explored, and not complex relationships between multiple features. Hence, the feature selection must be easily configurable. The parameters.yml file will allow the user to select X number of features, where X can be any number from 1 to the maximum amount of features in the dataset.

This allows us to easily test whether selecting features that we think are better will lead to better results, or whether leaving every feature in and letting the model figure it out would be better.

1. 'previously_contacted'. Extremely massive probability difference of subscribing depending on whether they have been previously contacted or not.
2. 'occupation/employed'. By far has the most interpretable correlations with the subscribed column as seen from bivariate analysis. However, only one of these should exist as they are highly correlated.
3. 'contact_method': Another easy to understand column with an obvious probability difference between the two values.
4. 'credit_default'. Also shows a clear split between the two values for which is more likely to lead to subscription. Is not ranked higher solely because the values of 'known' and 'unknown' are less interpretable unlike how clear 'occupation' and 'employed' are.
5. 'campaign_calls'. Shows a clear signal that after a certain amount of times calling the client, they will most likely not subscribe.
6. 'marital_status'. Shows that single people are more likely to be subscribed than those who are/were married. 'unknown' value is weird but not a problem as there were only 80 of them.
7. 'education_level'. Clear probability differences across values, but even more uninterpretable by human understanding and might be influenced by imbalanced data in categories.
8. 'age'. The individual correlation with the target starts to drop more from this feature onwards.
9. 'previous_contact_days'. It was helpful for getting the 'previously_contacted' feature, but does not show much valuable information otherwise.
10. 'housing_loan'. Very uninterpretable as the value with the biggest probability difference is 'unknown', with other values not showing much difference.
11. 'personal_loan'. Same issue as 'housing_loan', except worse.