main_notebook

June 3, 2021

1 Predicting Bank Telemarketing Sales

- Nick Gigliotti
- ndgigliotti@gmail.com

Table of Contents

2 Business Problem

Banco de Portugal has asked me to create a model to help them predict which customers are likely to invest in term deposit accounts as a result of telemarketing. Telemarketing is, no doubt, very stressful and time-consuming work. Salespersons don't like to waste the time of customers, because it's a waste of their time too. Not only that, but dealing with uninterested customers is surely the ugliest part the job. How many times a day does a bank telemarketer have to put up with insults and rude remarks? On the other hand, salespersons who are stuck calling low-potential customers are likely to resort to aggressive, desperate, sales tactics. It's like trench warfare over the phone, and it needs to be made easier.

That's where machine learning comes into play, and in particular **logistic regression**. Logistic regression models are widely used because they offer a good combination of simplicity and predictive power. My goal is to create a strong predictive model which can predict investments based on data which can be realistically obtained in advance. Banco de Portugal will use my model to increase the efficency of their telemarketing efforts by discovering the customers with the highest probability of investing.

3 Dataset

I train my predictive classifier on a Banco de Portugal telemarketing dataset which is publically available on the UCI Machine Learning Repository. The data was collected between May 2008 and November 2010.

[1]:
```
from distutils.util import strtobool
from functools import partial
from os.path import normpath

import matplotlib.pyplot as plt
import missingno as msno
import numpy as np
import pandas as pd
```

```
import seaborn as sns
from matplotlib import ticker

sns.set_theme(font_scale=1, style='darkgrid')
sns.set_palette("deep", desat=0.85, color_codes=True)
%matplotlib inline
```

[2]:
```
%load_ext autoreload
%autoreload 2
# My modules
from tools import cleaning, outliers, plotting, utils
from tools.modeling.classification import diagnostics

plt.rcParams.update(plotting.MPL_DEFAULTS)
```

There looks to be a mixture of categorical and numeric features. The feature labeled "y" is the target variable, namely whether or not the person invested in a term deposit.

[3]:
```
df = pd.read_csv(normpath("data/bank-additional-full.csv"), sep=";")
df.head()
```

[3]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	…	campaign	pdays	previous	poutcome	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	…	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	…	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
2	37	services	married	high.school	no	yes	no	telephone	may	mon	…	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	…	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
4	56	services	married	high.school	no	no	yes	telephone	may	mon	…	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no

[5 rows x 21 columns]

There are 21 features total and about 41k observations. About half of the features are "object" type, meaning that they're most likely categorical.

[4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             41188 non-null  int64
 1   job             41188 non-null  object
 2   marital         41188 non-null  object
 3   education       41188 non-null  object
 4   default         41188 non-null  object
 5   housing         41188 non-null  object
 6   loan            41188 non-null  object
 7   contact         41188 non-null  object
 8   month           41188 non-null  object
 9   day_of_week     41188 non-null  object
 10  duration        41188 non-null  int64
 11  campaign        41188 non-null  int64
 12  pdays           41188 non-null  int64
 13  previous        41188 non-null  int64
 14  poutcome        41188 non-null  object
 15  emp.var.rate    41188 non-null  float64
 16  cons.price.idx  41188 non-null  float64
 17  cons.conf.idx   41188 non-null  float64
 18  euribor3m       41188 non-null  float64
 19  nr.employed     41188 non-null  float64
 20  y               41188 non-null  object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
```

Yep, there are quite a few categorical variables. Even the numeric variables have strikingly few unique values for a dataset of 41k.

[5]:
```
df.nunique()
```

[5]:
```
age               78
job               12
marital            4
education          8
default            3
housing            3
loan               3
contact            2
month             10
day_of_week        5
duration        1544
campaign          42
pdays             27
previous           8
```

```
poutcome             3
emp.var.rate        10
cons.price.idx      26
cons.conf.idx       26
euribor3m          316
nr.employed         11
y                    2
dtype: int64
```

I rename some features to make them a little easier to interpret. Every variable prefixed with "contact" has to do with the last contact of the current campaign.

[6]:
```
df.columns = df.columns.str.replace(".", "_", regex=False)
rename = {"y": "invested",
          "poutcome": "prev_outcome",
          "pdays": "days_since_prev",
          "previous": "prev_contact_count",
          "campaign": "contact_count",
          "month": "contact_month",
          "day_of_week": "contact_weekday",
          "duration": "contact_duration",
          "contact": "contact_type",
          "nr_employed": "n_employed",
          "euribor3m": "euribor_3m"}
df.rename(columns=rename, inplace=True)
del rename
df.columns
```

[6]:
```
Index(['age', 'job', 'marital', 'education', 'default', 'housing', 'loan',
       'contact_type', 'contact_month', 'contact_weekday', 'contact_duration',
       'contact_count', 'days_since_prev', 'prev_contact_count',
       'prev_outcome', 'emp_var_rate', 'cons_price_idx', 'cons_conf_idx',
       'euribor_3m', 'n_employed', 'invested'],
      dtype='object')
```

3.1 Feature Explanations

Client Information

1. 'age' - years
2. 'job' - type of job
3. 'marital' - marital status
4. 'education' - level of education
5. 'default' - has defaulted on credit
6. 'housing' - has housing loan
7. 'loan' - has personal loan

Current Campaign

8. 'contact_type' - call type of **last contact** (cellular or landline)
9. 'contact_month' - month of **last contact**
10. 'contact_weekday' - weekday of **last contact**
11. 'contact_duration' - duration of **last contact** in seconds
12. 'contact_count' - total number of contacts during this campaign
13. 'invested' - invested in a term deposit (target variable)

    A term deposit is a short-term investment which typically matures within a few months or years.

Previous Campaigns

14. 'days_since_prev' - number of days since last contacted during previous campaign
15. 'prev_contact_count' - total number of contacts before this campaign
16. 'prev_outcome' - sales result of previous campaign

Economic Context

17. 'emp_var_rate' - employment variation rate (quarterly indicator)
18. 'cons_price_idx' - consumer price index (monthly indicator)
19. 'cons_conf_idx' - consumer confidence index (monthly indicator)
20. 'euribor_3m' - euribor 3 month rate (daily indicator)
21. 'n_employed' - thousands of people employed (quarterly indicator)

4 Initial Cleaning

I do some preliminary tidying up and reorganization but leave most of the preprocessing for the modeling section. Using Sklearn's preprocessing pipelines allows the preprocessors and their parameters to be adjusted alongside the model itself.

I begin by replacing some placeholders with NaN and surveying the missing values and duplicates. There are 12 duplicate rows—an easy number to drop.

[7]:
```
df["days_since_prev"].replace(999, np.NaN, inplace=True)
df.replace(["unknown", "nonexistent"], np.NaN, inplace=True)
cleaning.info(df)
```

[7]:

	nan	nan_%	uniq	uniq_%	dup	dup_%
days_since_prev	39673	96.32	26	0.06	12	0.03
prev_outcome	35563	86.34	2	0.00	12	0.03
default	8597	20.87	2	0.00	12	0.03
education	1731	4.20	7	0.02	12	0.03
housing	990	2.40	2	0.00	12	0.03
loan	990	2.40	2	0.00	12	0.03
job	330	0.80	11	0.03	12	0.03
marital	80	0.19	3	0.01	12	0.03
age	0	0.00	78	0.19	12	0.03
n_employed	0	0.00	11	0.03	12	0.03
euribor_3m	0	0.00	316	0.77	12	0.03

```
cons_conf_idx           0   0.00    26    0.06   12   0.03
cons_price_idx          0   0.00    26    0.06   12   0.03
emp_var_rate            0   0.00    10    0.02   12   0.03
contact_duration        0   0.00  1544    3.75   12   0.03
prev_contact_count      0   0.00     8    0.02   12   0.03
contact_count           0   0.00    42    0.10   12   0.03
contact_weekday         0   0.00     5    0.01   12   0.03
contact_month           0   0.00    10    0.02   12   0.03
contact_type            0   0.00     2    0.00   12   0.03
invested                0   0.00     2    0.00   12   0.03
```

I drop the duplicate rows.

[8]:
```
display(df.loc[df.duplicated()])
df.drop_duplicates(inplace=True)
```

```
       age          job   marital            education default housing loan  \
1266    39  blue-collar   married             basic.6y      no      no   no
12261   36      retired   married                  NaN      no      no   no
14234   27   technician    single  professional.course      no      no   no
16956   47   technician  divorced          high.school      no     yes   no
18465   32   technician    single  professional.course      no     yes   no
20216   55     services   married          high.school     NaN      no   no
20534   41   technician   married  professional.course      no     yes   no
25217   39       admin.   married    university.degree      no      no   no
28477   24     services    single          high.school      no     yes   no
32516   35       admin.   married    university.degree      no     yes   no
36951   45       admin.   married    university.degree      no      no   no
38281   71      retired    single    university.degree      no      no   no

      contact_type contact_month contact_weekday  …  contact_count  \
1266     telephone           may             thu  …              1
12261    telephone           jul             thu  …              1
14234     cellular           jul             mon  …              2
16956     cellular           jul             thu  …              3
18465     cellular           jul             thu  …              1
20216     cellular           aug             mon  …              1
20534     cellular           aug             tue  …              1
25217     cellular           nov             tue  …              2
28477     cellular           apr             tue  …              1
32516     cellular           may             fri  …              4
36951     cellular           jul             thu  …              1
38281    telephone           oct             tue  …              1

       days_since_prev  prev_contact_count  prev_outcome emp_var_rate  \
1266               NaN                   0           NaN          1.1
12261              NaN                   0           NaN          1.4
14234              NaN                   0           NaN          1.4
```

```
16956              NaN                   0           NaN          1.4
18465              NaN                   0           NaN          1.4
20216              NaN                   0           NaN          1.4
20534              NaN                   0           NaN          1.4
25217              NaN                   0           NaN         -0.1
28477              NaN                   0           NaN         -1.8
32516              NaN                   0           NaN         -1.8
36951              NaN                   0           NaN         -2.9
38281              NaN                   0           NaN         -3.4

       cons_price_idx  cons_conf_idx  euribor_3m  n_employed  invested
1266           93.994          -36.4       4.855      5191.0        no
12261          93.918          -42.7       4.966      5228.1        no
14234          93.918          -42.7       4.962      5228.1        no
16956          93.918          -42.7       4.962      5228.1        no
18465          93.918          -42.7       4.968      5228.1        no
20216          93.444          -36.1       4.965      5228.1        no
20534          93.444          -36.1       4.966      5228.1        no
25217          93.200          -42.0       4.153      5195.8        no
28477          93.075          -47.1       1.423      5099.1        no
32516          92.893          -46.2       1.313      5099.1        no
36951          92.469          -33.6       1.072      5076.2       yes
38281          92.431          -26.9       0.742      5017.5        no

[12 rows x 21 columns]
```

Now I survey the uniques to see how I should process the categorical variables.

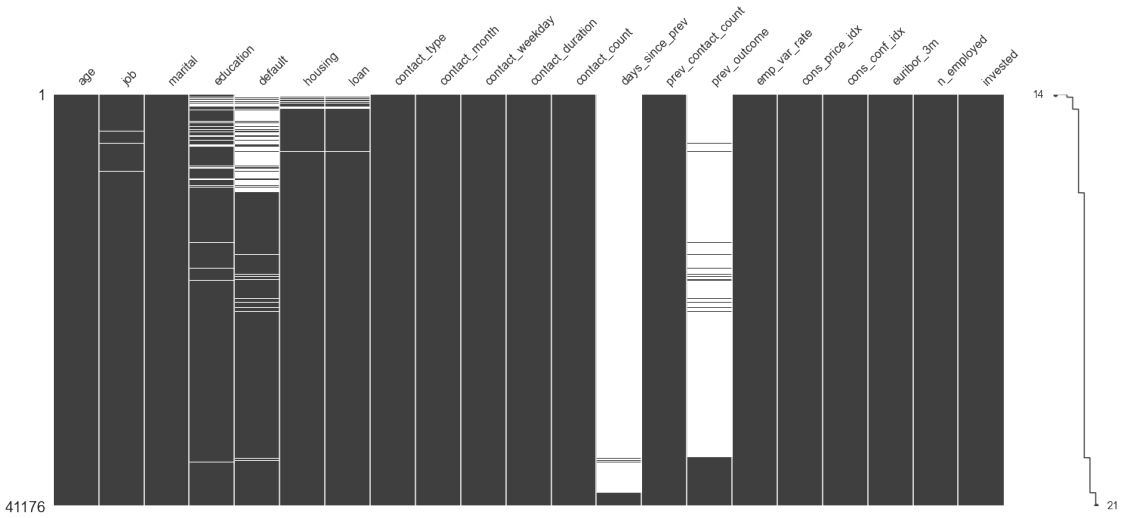
[9]:
```
cleaning.show_uniques(df, cut=20)
```

```
<IPython.core.display.HTML object>
```

Doesn't look like there are any interesting patterns with the missing values. It's striking how empty 'days_since_prev' is, and that it doesn't match up with 'prev_outcome'.

[10]:
```
msno.matrix(df, sort="ascending")
```

[10]:
```
<AxesSubplot:>
```



I go ahead and encode 'invested' as numeric for convenience. It's the prediction target and I'll need to make calculations for EDA. The classes are imbalanced, which is a problem I'll have to deal with in the modeling phase.

[11]:
```
df["invested"] = (df["invested"] == "yes").astype(np.float64)
df["invested"].value_counts(normalize=True)
```

[11]:
```
0.0    0.887337
1.0    0.112663
Name: invested, dtype: float64
```

I one-hot encode 'prev_outcome' and implicitly drop the null category. This eliminates a lot of null values and avoids multicollinearity at the same time.

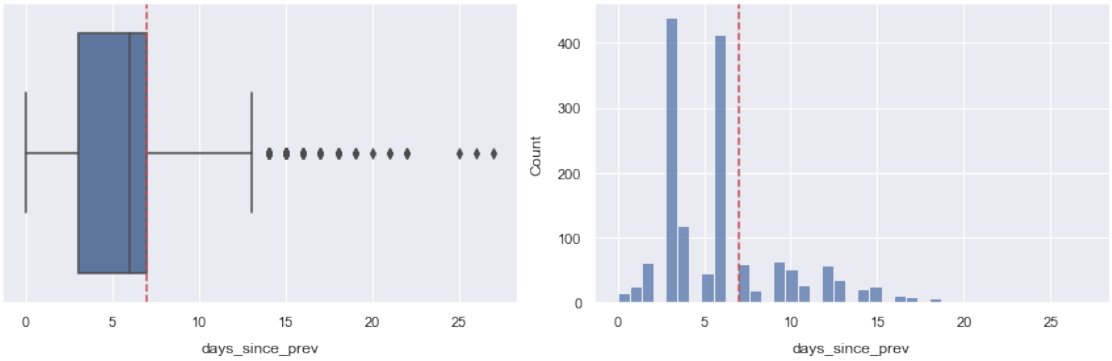
Nevertheless, the categories are wildly uneven. It isn't the target variable, but it still isn't good. Most of the customers in the dataset weren't contacted during a previous campaign, so those who were are a rare minority.

[12]:
```
df = df.join(pd.get_dummies(df["prev_outcome"], prefix="prev"))

df[["prev_failure", "prev_success"]].value_counts(normalize=True)
```

[12]:
```
prev_failure  prev_success
0             0              0.863391
1             0              0.103264
0             1              0.033345
dtype: float64
```

One week seems like a good cutoff for turning "days_since_prev" into a categorical. The third quartile is exactly 7 days.

[13]:
```
# make subplots
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 4))

# plot boxplot and hist
sns.boxplot(data=df, x="days_since_prev", ax=ax1)
sns.histplot(data=df, x="days_since_prev", ax=ax2)

# add line at 7 days
for ax in [ax1, ax2]:
    ax.axvline(7, c="r", ls="--")

fig.tight_layout()

# mop up for performance
del fig, ax1, ax2
```



I create a "recent_prev_contact" feature to replace the "days_since_prev" feature with 96% nulls. I let the nulls go False, although now the categories are just wildly uneven.

[14]:
```
df["recent_prev_contact"] = df["days_since_prev"] <= 7
df["recent_prev_contact"].value_counts(1)
```

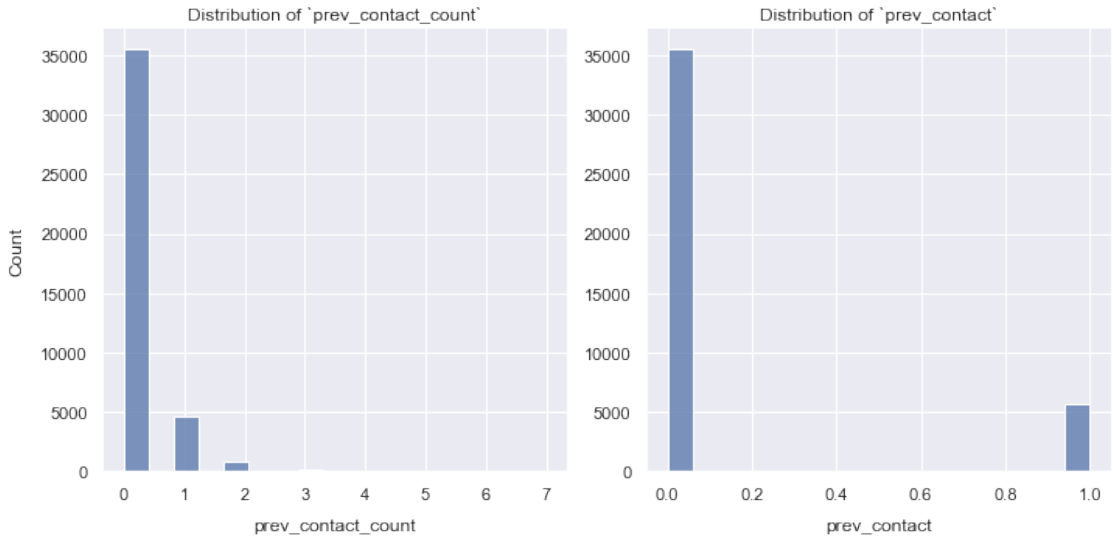
[14]:
```
False    0.971415
True     0.028585
Name: recent_prev_contact, dtype: float64
```

I turn "prev_contact_count" into a binary categorical because values over 1 are ruled outliers by the z-score and Tukey fence methods. A 90% Winsorization results in a nice clean binary feature. The new feature simply indicates whether or not the customer was contacted during a previous campaign.

[15]:
```
# squeeze values to central 90%
df["prev_contact"] = outliers.quantile_winsorize(
    df["prev_contact_count"], inner=0.9)
```

```
plotting.multi_dist(data=df[["prev_contact_count", "prev_contact"]]);
```

```
                    n_winsorized  pct_winsorized
prev_contact_count          1064         2.58403
total_obs                   1064         2.58403
```



I convert the binary "contact_type" feature to a boolean feature "contact_cellular". Again, every variable with the "contact" prefix has to do with the last contact of the current campaign.

[16]:
```
df["contact_cellular"] = df["contact_type"] == "cellular"
df["contact_cellular"].value_counts()
```

[16]:
```
True     26135
False    15041
Name: contact_cellular, dtype: int64
```

I drop the features which I've redesigned.

[17]:
```
to_drop = ["prev_outcome", "days_since_prev",
           "prev_contact_count", "contact_type"]
df.drop(to_drop, axis=1, inplace=True)
df.columns
```

[17]:
```
Index(['age', 'job', 'marital', 'education', 'default', 'housing', 'loan',
       'contact_month', 'contact_weekday', 'contact_duration', 'contact_count',
       'emp_var_rate', 'cons_price_idx', 'cons_conf_idx', 'euribor_3m',
       'n_employed', 'invested', 'prev_failure', 'prev_success',
       'recent_prev_contact', 'prev_contact', 'contact_cellular'],
      dtype='object')
```

I go ahead and numerically encode these binary string categoricals, preserving NaNs.

[18]:
```
string_cols = ["default", "housing", "loan"]
df[string_cols] = df[string_cols].applymap(strtobool, "ignore")
cleaning.show_uniques(df, columns=string_cols)
del string_cols
```

```
<IPython.core.display.HTML object>
```

For now I will hold off on converting these binary variables to categorical dtype. It will be easier to work with them as numeric variables, since they don't need to be one-hot encoded.

[19]:
```
binary_cats = utils.binary_cols(df)
df[binary_cats] = df[binary_cats].astype(np.float64)
cleaning.show_uniques(df, columns=binary_cats)
```

```
<IPython.core.display.HTML object>
```

Looks like most of the binary categoricals are imbalanced. By far the worst is 'default'.

[20]:
```
cleaning.token_info(df[binary_cats], normalize=True)
```

[20]:

	min_tokens	max_tokens	types
default	0.000092	0.999908	2.0
recent_prev_contact	0.028585	0.971415	2.0
prev_success	0.033345	0.966655	2.0
prev_failure	0.103264	0.896736	2.0
invested	0.112663	0.887337	2.0
prev_contact	0.136609	0.863391	2.0
loan	0.155477	0.844523	2.0
contact_cellular	0.365286	0.634714	2.0
housing	0.463221	0.536779	2.0

I drop the most extremely uneven binary category: 'default'. It has almost no True values, and these are essentially outliers. I'm not interested in the rare clients who have defaulted on credit or a loan.

[21]:
```
df.drop(columns=["default"], inplace=True)
del binary_cats
cleaning.token_info(df.loc[:, df.nunique() == 2], normalize=True)
```

[21]:

	min_tokens	max_tokens	types
recent_prev_contact	0.028585	0.971415	2.0
prev_success	0.033345	0.966655	2.0
prev_failure	0.103264	0.896736	2.0
invested	0.112663	0.887337	2.0
prev_contact	0.136609	0.863391	2.0
loan	0.155477	0.844523	2.0
contact_cellular	0.365286	0.634714	2.0
housing	0.463221	0.536779	2.0

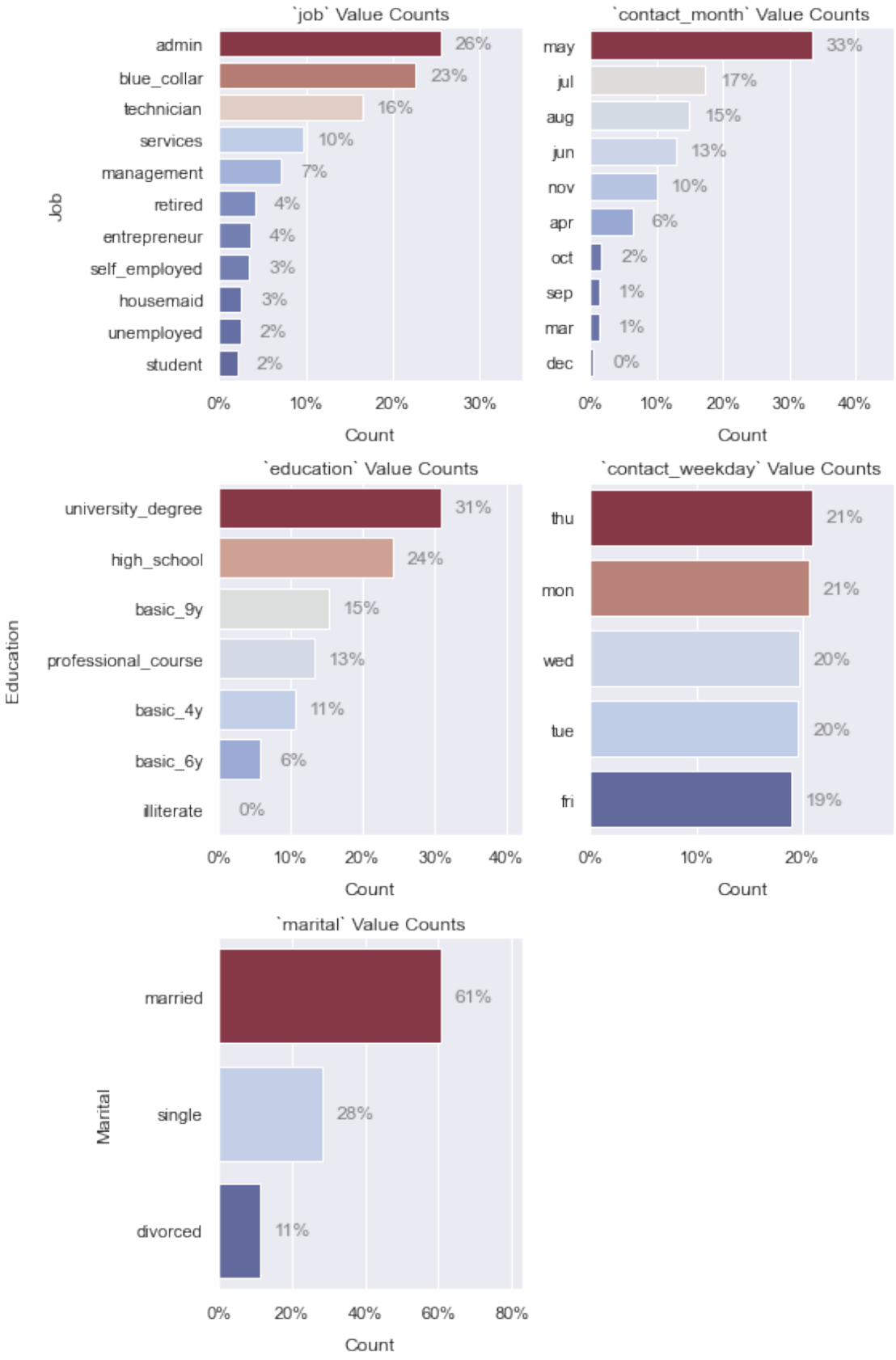
I tidy up the labels for variables with 2+ categories.

[22]:
```
multi_cat = ["job", "marital", "education",
             "contact_month", "contact_weekday"]

# tweak some labels
df["job"] = df["job"].str.replace(".", "", regex=False)
df["job"] = df["job"].str.replace("-", "_", regex=False)
df["education"] = df["education"].str.replace(".", "_", regex=False)

# convert to unordered categoricals
df[multi_cat] = df[multi_cat].astype("category")


cleaning.show_uniques(df, columns=multi_cat)
```

```
<IPython.core.display.HTML object>
```

The distributions look serviceable, but there are a couple extremely thin categories (under 1%): “dec” and “illiterate”.

[23]:
```
plotting.multi_countplot(data=df[multi_cat],
                         normalize=True,
                         ncols=2,
                         height=4)
del multi_cat
```


`job` Value Counts
admin 26%
blue_collar 23%
technician 16%
services 10%
management 7%
retired 4%
entrepreneur 4%
self_employed 3%
housemaid 3%
unemployed 2%
student 2%
Job
0% 10% 20% 30%
Count
`contact_month` Value Counts
may 33%
jul 17%
aug 15%
jun 13%
nov 10%
apr 6%
oct 2%
sep 1%
mar 1%
dec 0%
0% 10% 20% 30% 40%
Count
`education` Value Counts
university_degree 31%
high_school 24%
basic_9y 15%
professional_course 13%
basic_4y 11%
basic_6y 6%
illiterate 0%
Education
0% 10% 20% 30% 40%
Count
`contact_weekday` Value Counts
thu 21%
mon 21%
wed 20%
tue 20%
fri 19%
0% 10% 20%
Count
`marital` Value Counts
married 61%
single 28%
divorced 11%
Marital
0% 20% 40% 60% 80%
Count

I drop the extremely thin "illiterate" and "dec" classes. With too few examples to accurately represent a real pattern, these will likely just add noise to the model. Plus illiterate people are a just a rare minority to begin with, and not really of any interest. Perhaps if this were a Hooked on Phonics™ dataset…

[24]:
```
# compute rows to keep
keep = (df.education != "illiterate") & (df.contact_month != "dec")

# overwrite `df` with keeper rows
df = df.loc[keep].copy()

# drop unused categories
df["education"] = df["education"].cat.remove_unused_categories()
df["contact_month"] = df["contact_month"].cat.remove_unused_categories()

# view results
print(f"Dropped {(~keep).sum()} observations.")
del keep
cleaning.token_info(df[["education", "contact_month"]], normalize=True)
```

```
Dropped 200 observations.
```

[24]:

	min_tokens	max_tokens	types
contact_month	0.013325	0.335904	9.0
education	0.058355	0.308049	6.0

I order the weekdays and months for plotting purposes.

[25]:
```
# define order
days = ["mon", "tue", "wed", "thu", "fri"]
months = ["mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov"]

# convert to ordered categories
df["contact_weekday"].cat.reorder_categories(days, ordered=True, inplace=True)
df["contact_month"].cat.reorder_categories(months, ordered=True, inplace=True)

# mop up temp variables
del days, months

display(df["contact_weekday"].cat.categories)
display(df["contact_month"].cat.categories)
```

```
Index(['mon', 'tue', 'wed', 'thu', 'fri'], dtype='object')
```

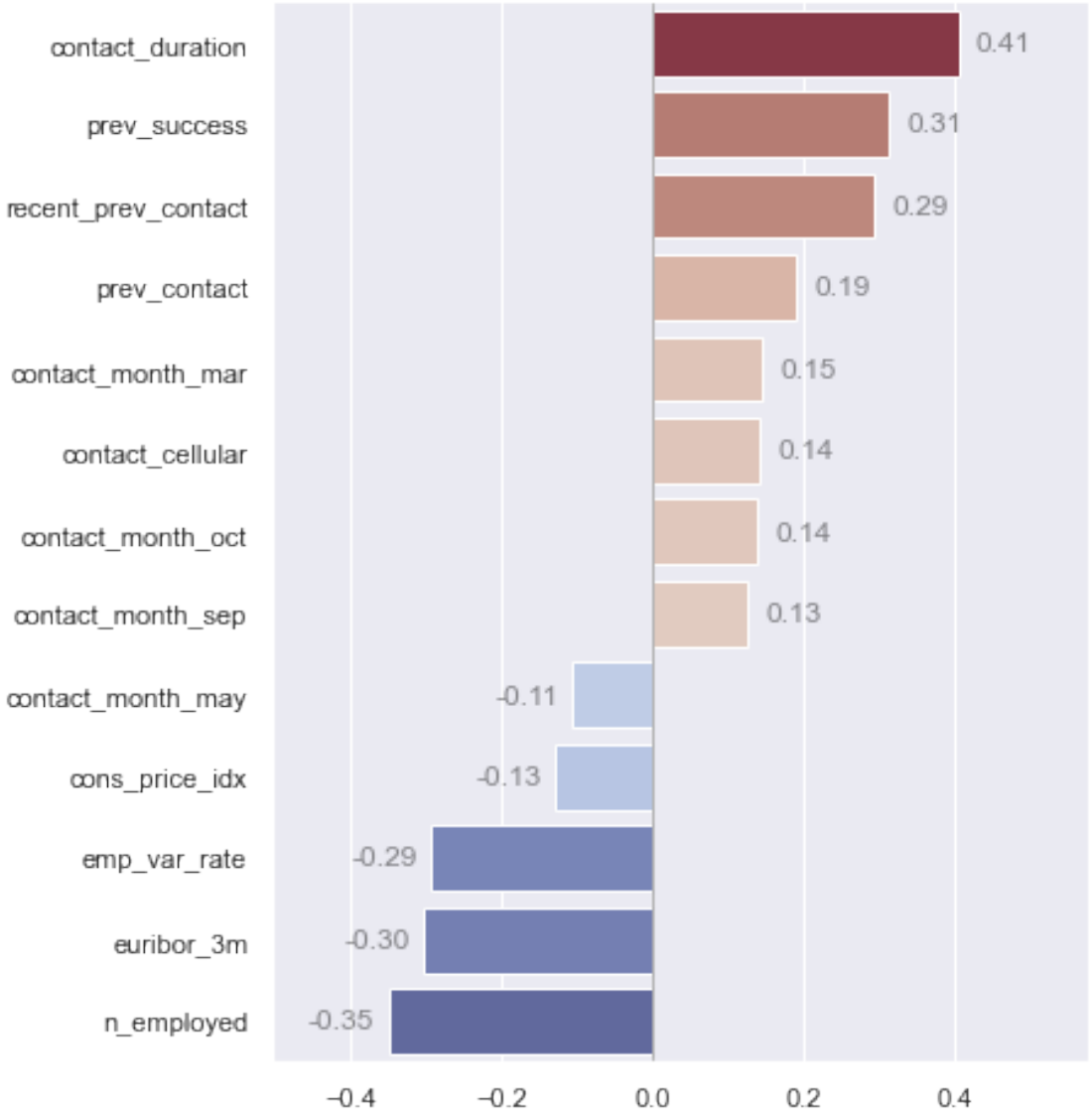
```
Index(['mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov'],␣
↪dtype='object')
```

5 Exploration

Here are correlations between numeric/boolean variables and the target. As stated in the description on the UCI page, 'contact_duration' has a strong relationship with 'invested'. I will ignore this feature later on, because it's not information which could be obtained in advance.

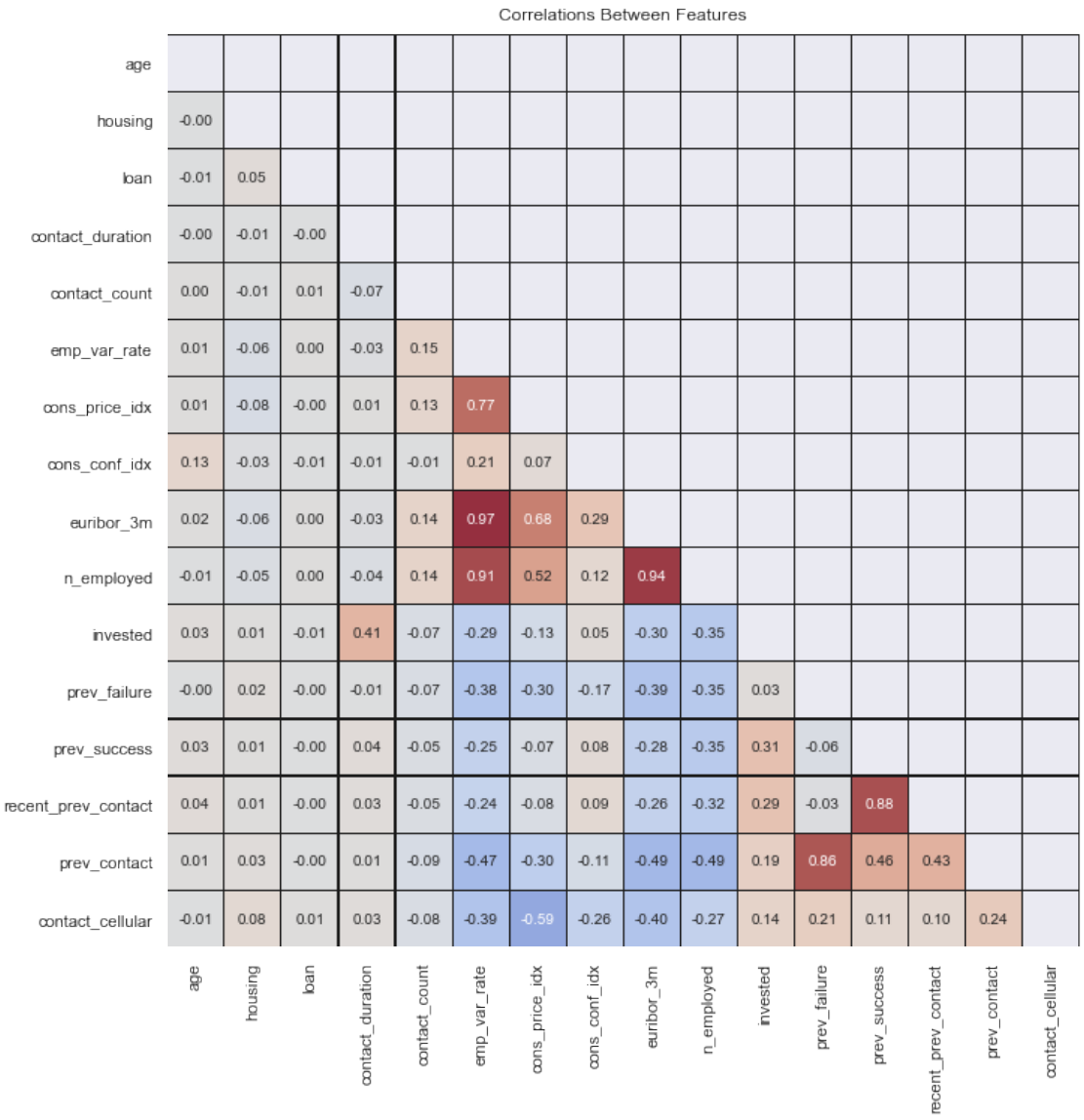
Unsurprisingly, 'prev_success' and 'recent_prev_contact' have strong relationships with the target. Strangely, 'n_employed' has a negative relationship with the target, meaning that people tend to invest when fewer people are employed.

[26]:
```
inv_corr = pd.get_dummies(df.drop(columns="invested"))
inv_corr = inv_corr.corrwith(df["invested"])
inv_corr = inv_corr.loc[inv_corr.abs() > .1]
ax = plotting.heated_barplot(data=inv_corr)
plotting.annot_bars(ax)
del inv_corr
```



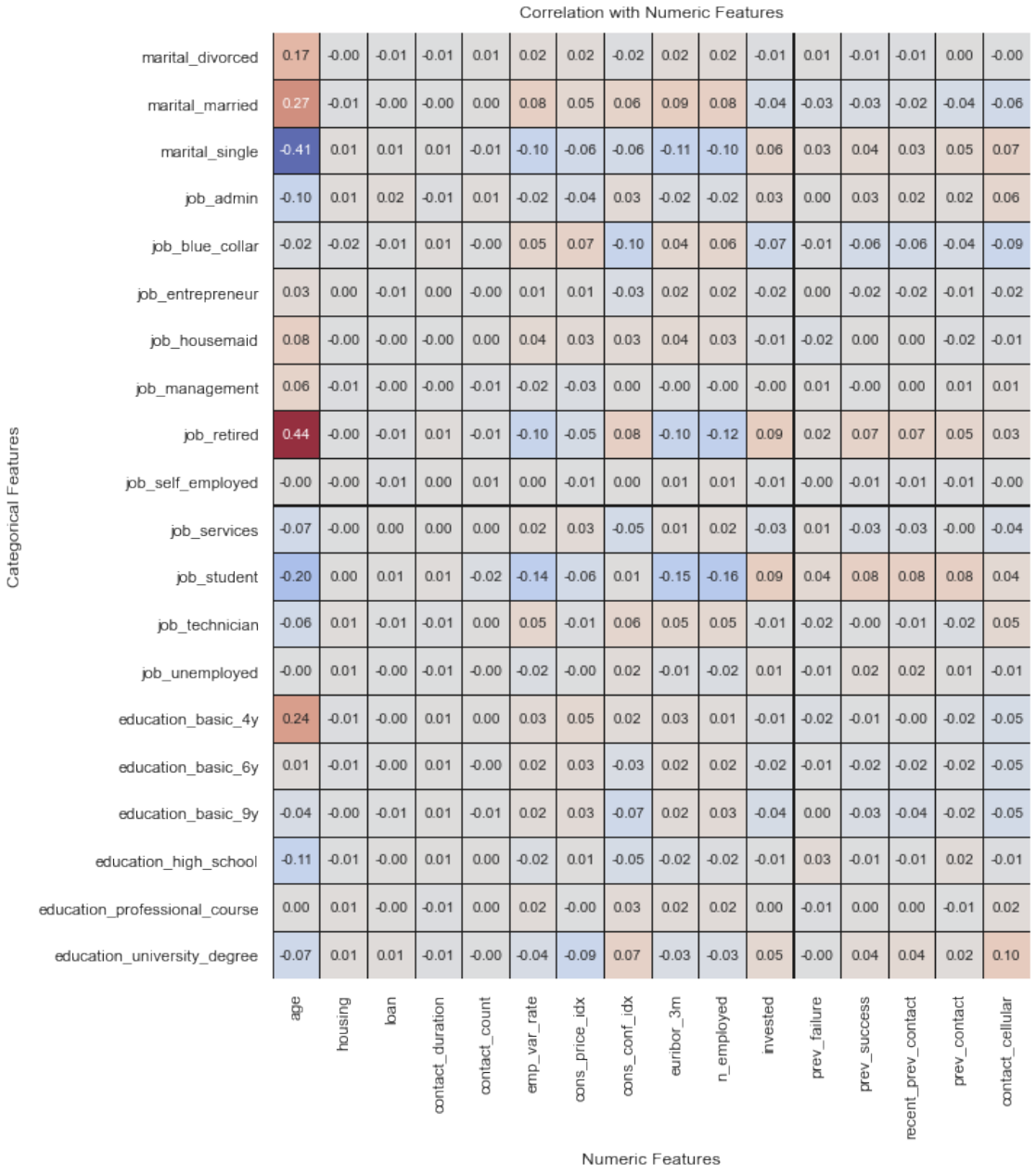
[27]:
```
plotting.pair_corr_heatmap(data=df, scale=.7)
```

[27]: <AxesSubplot:title={'center':'Correlations Between Features'}>

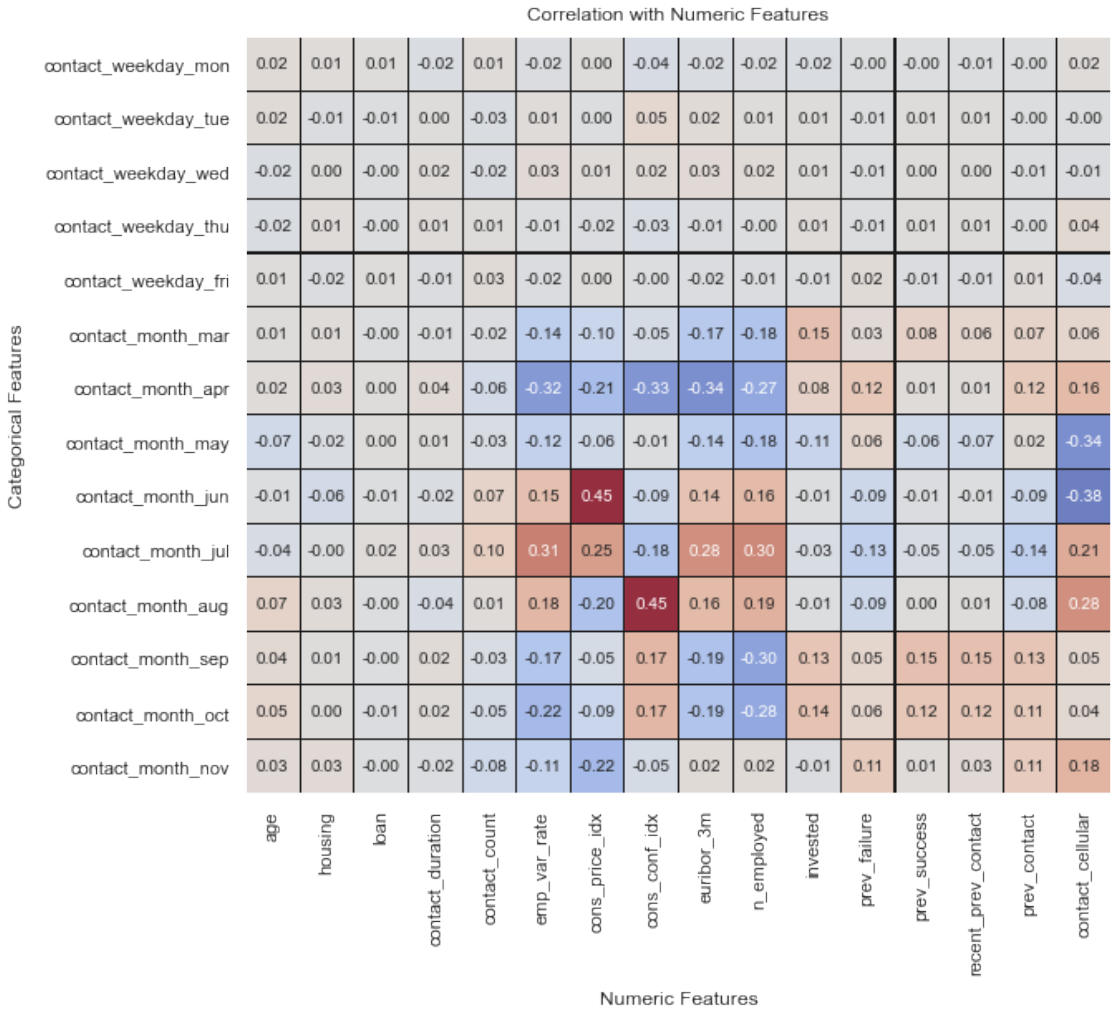


[28]:
```
plotting.cat_corr_heatmap(data=df,
                          categorical=["marital", "job", "education"],
                          scale=.6,
                          fmt=".2f")
```

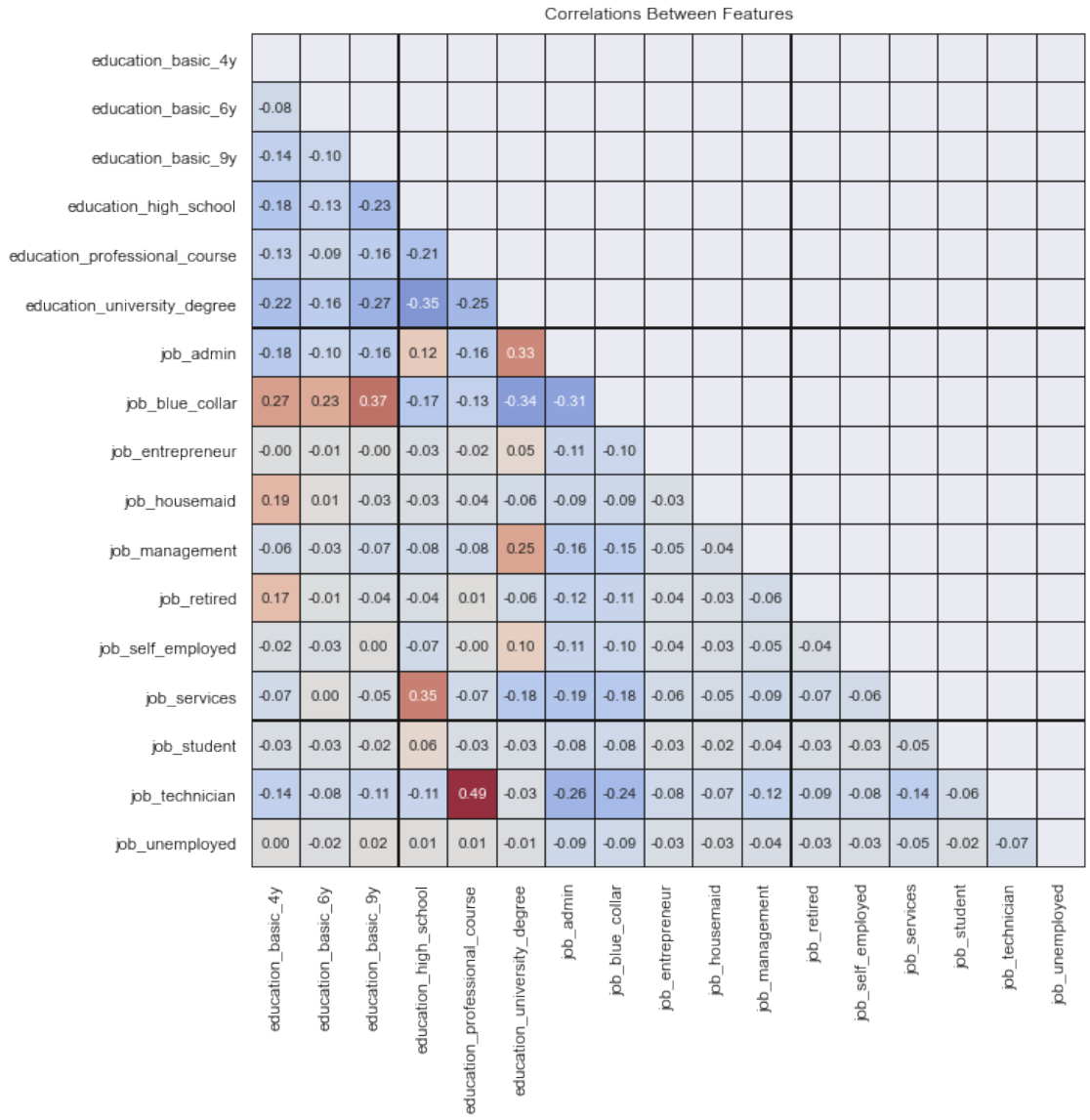
[28]: <AxesSubplot:title={'center':'Correlation with Numeric Features'},
xlabel='Numeric Features', ylabel='Categorical Features'>



[29]:
```
temporal = ["contact_weekday", "contact_month"]
plotting.cat_corr_heatmap(data=df, categorical=temporal, scale=.6, fmt=".2f")
del temporal
```

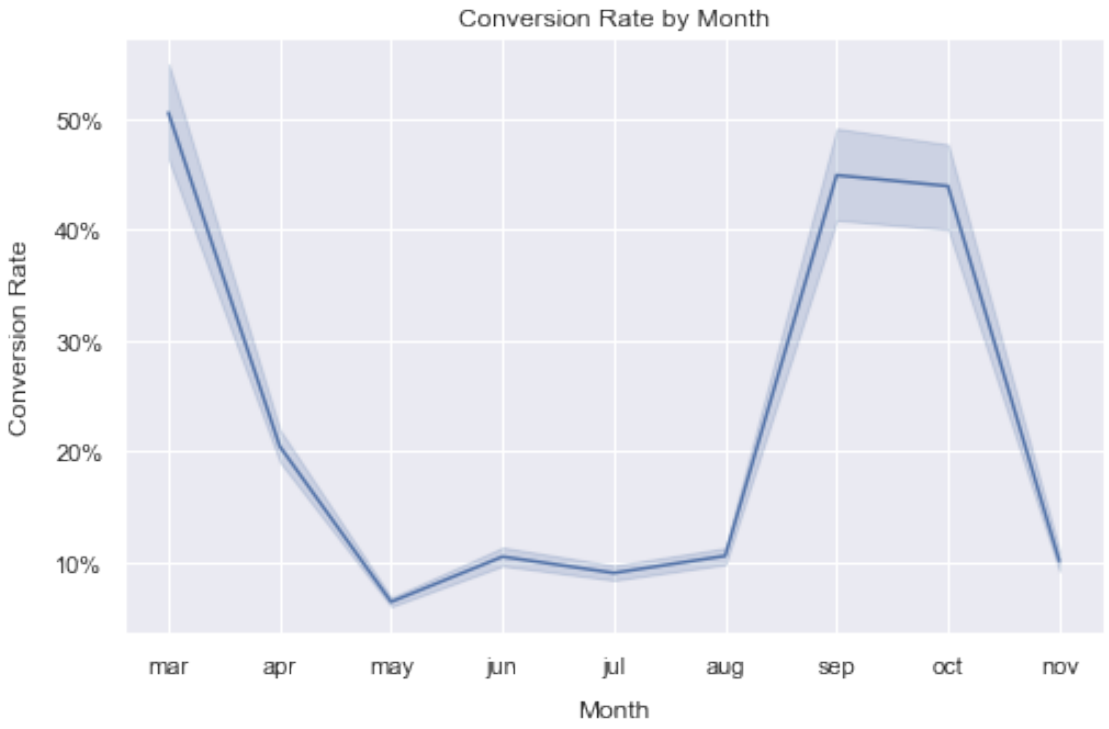


[30]:
```
dummies = pd.get_dummies(df[["education", "job"]])
plotting.pair_corr_heatmap(data=dummies, scale=.6, fmt=".2f")
del dummies
```



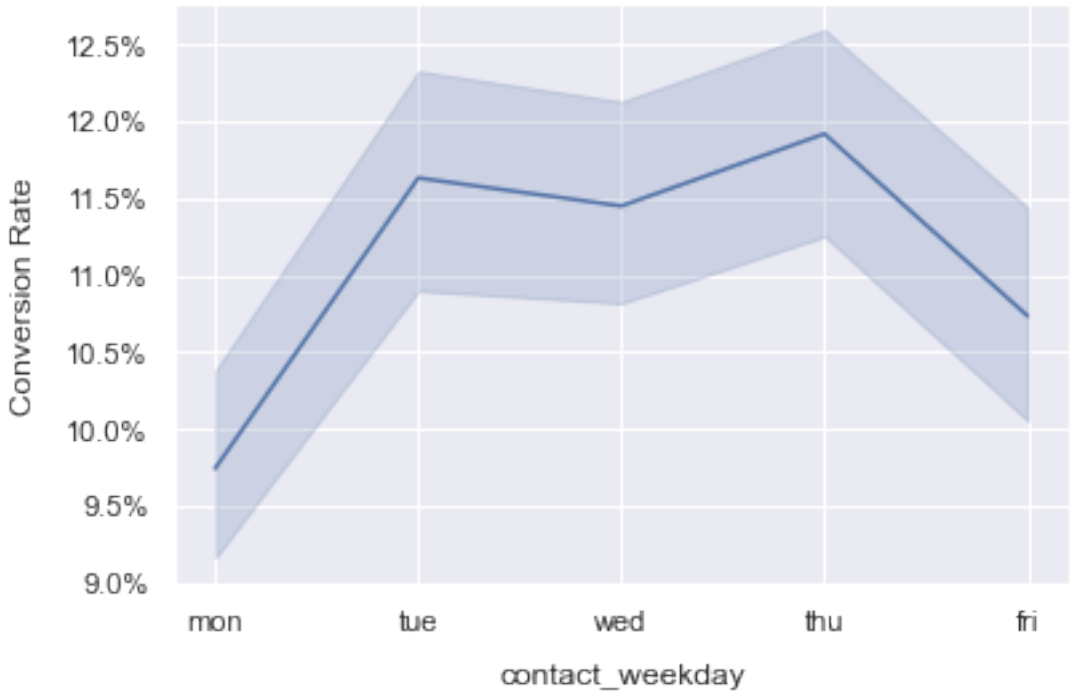
[31]:
```
fig, ax = plt.subplots(figsize=(8, 5))
sns.lineplot(data=df, x="contact_month", y="invested", ax=ax)
ax.yaxis.set_major_formatter(ticker.PercentFormatter(xmax=1))
ax.set(title="Conversion Rate by Month", ylabel="Conversion Rate",␣
↪xlabel="Month")
```

[31]:
```
[Text(0.5, 1.0, 'Conversion Rate by Month'),
 Text(0, 0.5, 'Conversion Rate'),
 Text(0.5, 0, 'Month')]
```



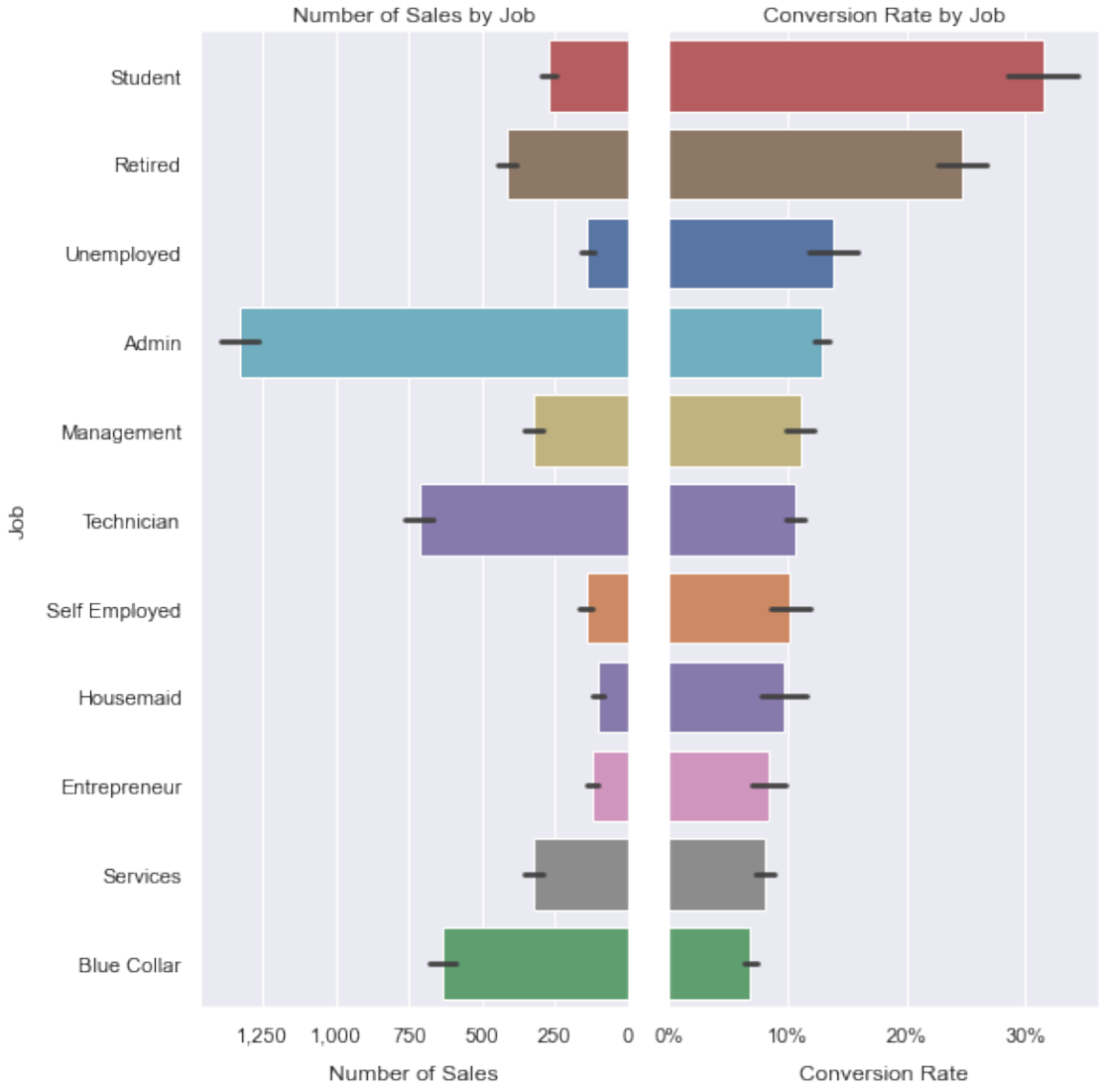
[32]:
```
ax = sns.lineplot(data=df, x="contact_weekday", y="invested")
ax.yaxis.set_major_formatter(ticker.PercentFormatter(xmax=1, decimals=1))
ax.set_ylabel("Conversion Rate")
```

[32]:
```
Text(0, 0.5, 'Conversion Rate')
```

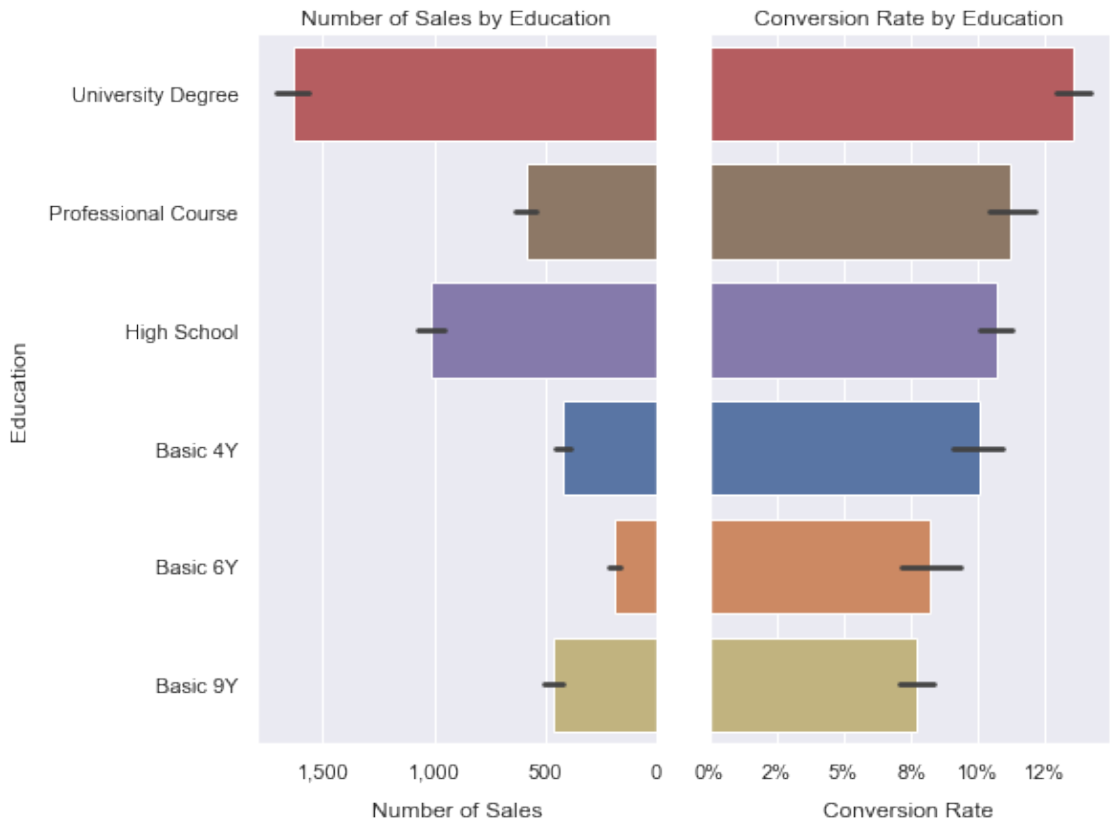


[33]:
```python
def conversion_bars(*, data, y,  x="invested", **kwargs):
    """Draw mirror plot of sales counts and conversion rates."""

    # Format strings for pretty titles
    x, y = utils.to_title(x), utils.to_title(y)
    data = utils.title_mode(data)

    # Plot mirror plot
    fig = plotting.mirror_plot(data=data, x=x, y=y, **kwargs)

    # Format tick labels
    ax2, ax1 = fig.get_axes()
    ax1.xaxis.set_major_formatter(ticker.PercentFormatter(xmax=1, decimals=0))
    ax2.xaxis.set_major_formatter(ticker.StrMethodFormatter("{x:,.0f}"))

    # Label axes
    ax1.set(xlabel="Conversion Rate",
            ylabel=None,
            title=f"Conversion Rate by {y}")
    ax2.set(xlabel="Number of Sales",
            ylabel=y,
            title=f"Number of Sales by {y}")

    return fig
```

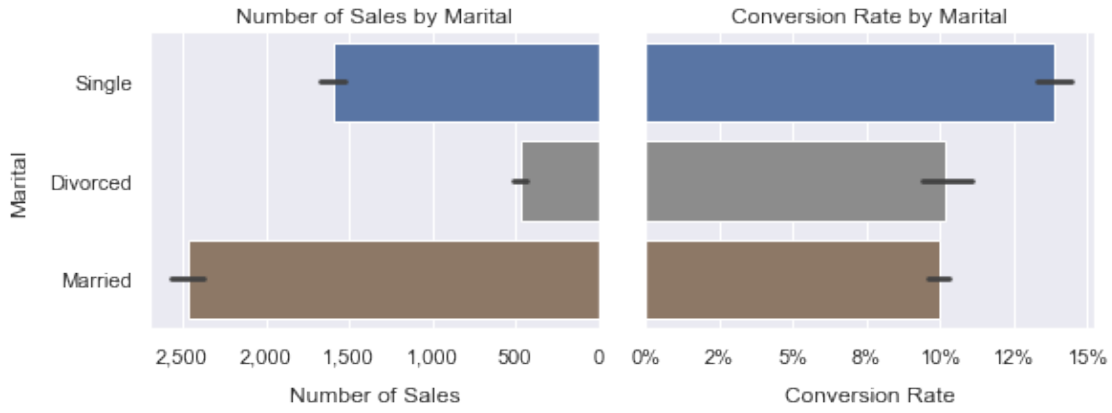
[34]:
```
conversion_bars(data=df, y="job", size=(4,8));
```



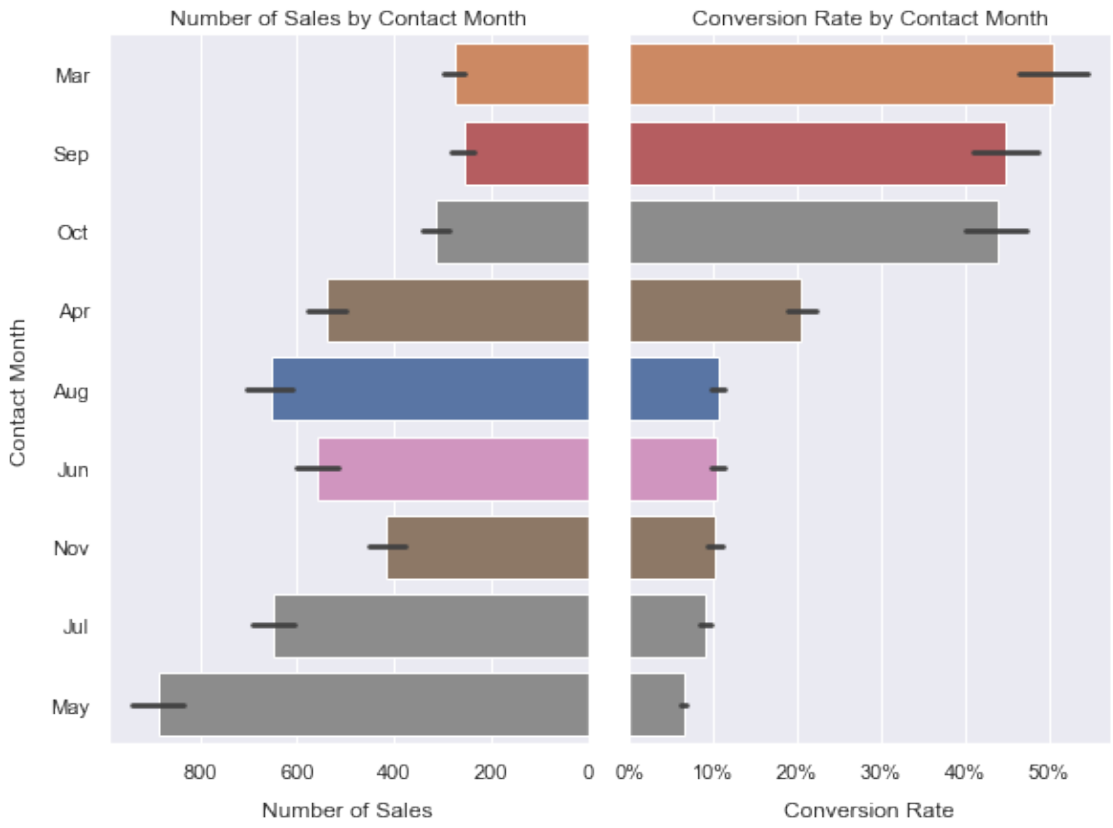
[35]:
```
conversion_bars(data=df, x="invested", y="education", size=(4, 6));
```



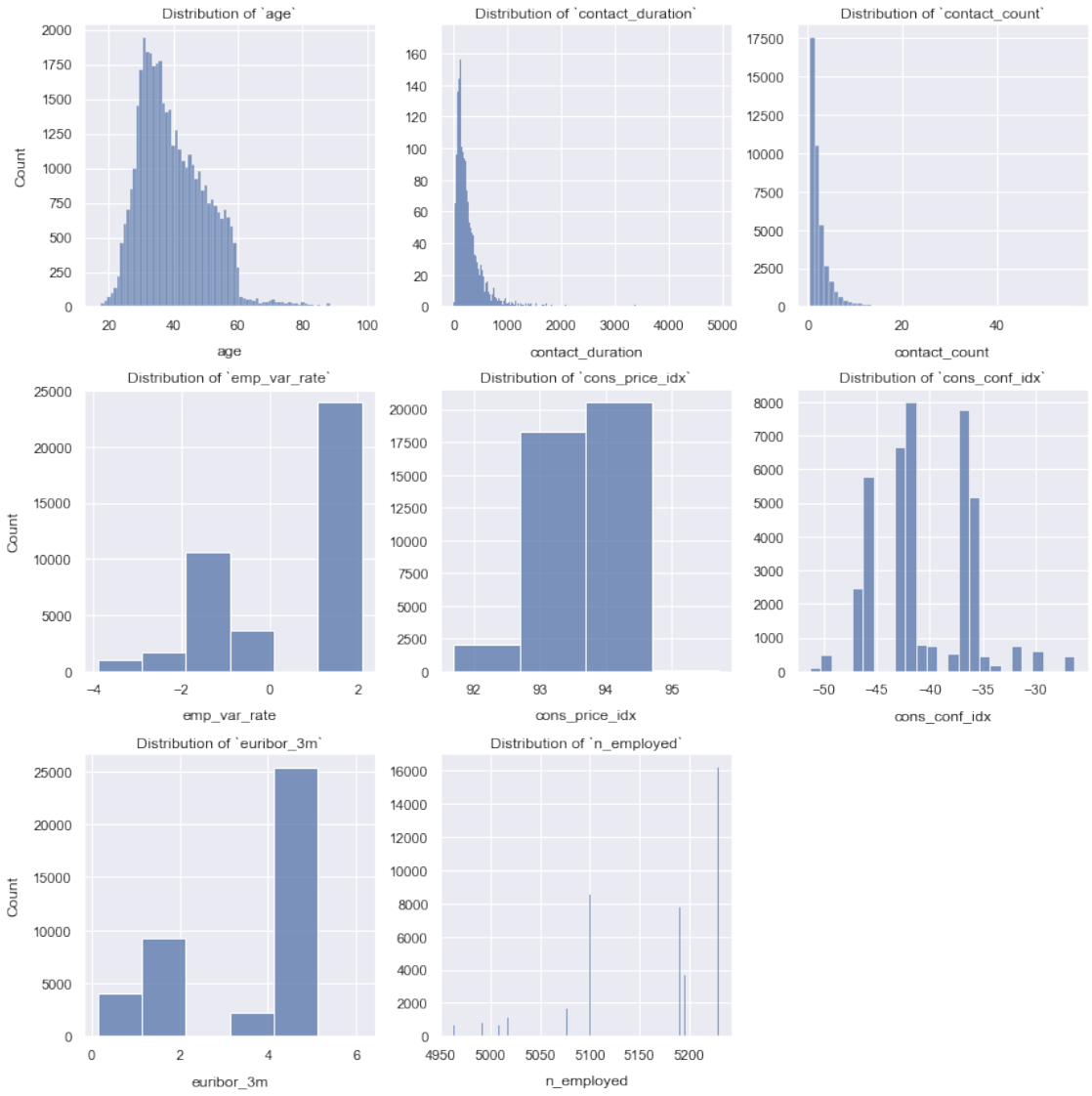
[36]:
```
conversion_bars(data=df, y="marital", size=(4, 3));
```



[37]:
```
conversion_bars(data=df, y="contact_month", size=(4, 6));
```

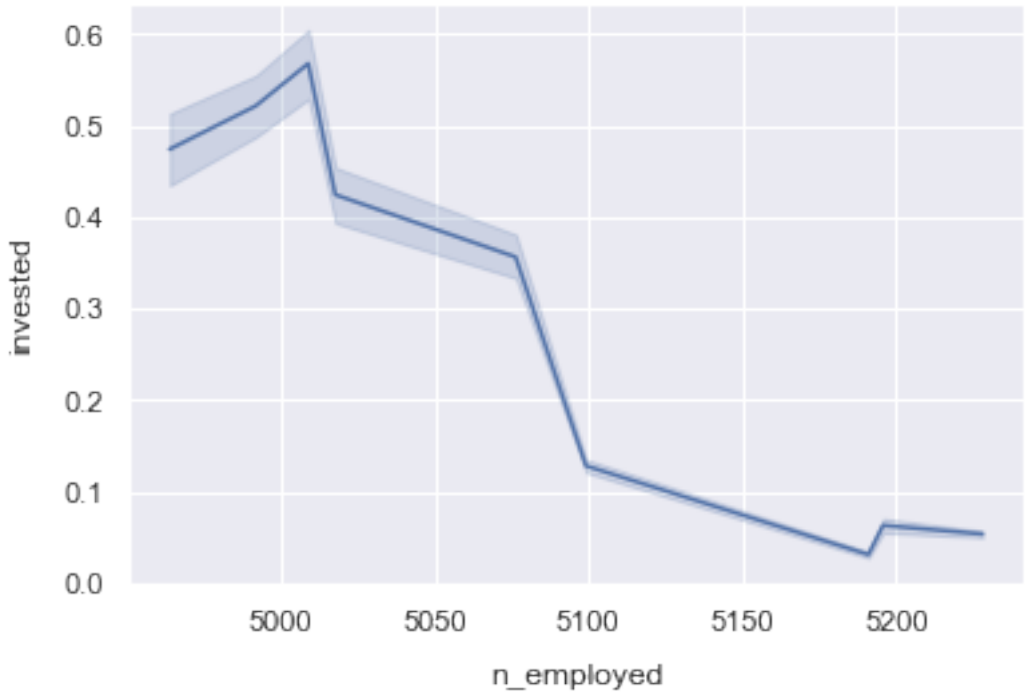


[38]:
```
plotting.multi_dist(data=df[utils.true_numeric_cols(df)],
                    discrete=True,
                    height=4);
```



[39]:
```
sns.lineplot(data=df, x="n_employed", y="invested")
```

[39]:
```
<AxesSubplot:xlabel='n_employed', ylabel='invested'>
```



[ ]:

6 Modeling

6.0.1 Modeling Imports

[75]:
```
import joblib
from feature_engine.outliers import Winsorizer
from feature_engine.selection import SmartCorrelatedSelection
from sklearn.base import clone
from sklearn.dummy import DummyClassifier
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (GridSearchCV, RepeatedKFold,
                                     train_test_split)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler
```

My classes/modules:

[41]:
```
from tools.modeling.preprocessing import DummyEncoder
from tools.modeling import selection
```

6.0.2 Train-test Split

I begin my iterative modeling process by performing a train-test split.

I drop "contact_duration", because as noted in on the UCI Repo for this dataset, it reduces the practical value of the model. This was the duration of the last call, after which the broker knew whether or not the customer invested. The duration of the final call wouldn't be known prior to the final call. Since this is not data that Banco de Portugal would have to **plug into** my predictive model, so there's no point in including it. It does, however, make me sad to exclude it, because it radically improves the performance of the model.

[42]:
```
# drop NaNs and irrelevant columns
X = df.drop(columns=["invested", "contact_duration"])

# drop NaNs and slice target column
y = df["invested"]

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[42]:
```
((30732, 19), (10244, 19), (30732,), (10244,))
```

6.1 First Model

For my baseline model, I apply minimal preprocessing to the data.

6.1.1 Baseline Preprocessors

DummyEncoder I one-hot encode the categorical variables using `DummyEncoder`, which is my wrapper for `pd.get_dummies`. It conveniently stores all the feature names, unlike the scikit `OneHotEncoder`.

SmartCorrelatedSelection I use a feature selection tool which detects highly correlated sets of features and then drops all of them except one, which it chooses intelligently. The purpose is to avoid multi-collinearity, which is a potential problem in logistic regression. If features are highly inter-correlated, the model may have trouble distinguishing between their individual influences, and the coefficients may be distorted. I have it set to keep the feature with the highest variance, which is a simple, fast, and reasonably effective strategy.

SimpleImputer Due to my data cleaning efforts, there are not very many missing values left in the dataset. The missing values for multi-category categorical variables like 'education', 'job', and 'marital' will turn to 0.0 (i.e. False) during one-hot encoding. There is not really a more sophisticated way to handle those, except perhaps filling with mode. Since I have already rendered other missing categorical values as 0.0, as when I created 'prev_failure' and 'prev_success', I continue this policy here. I use `SimpleImputer` to fill missing values with 0.0 in the remaining two binary categoricals, 'housing' and 'loan'. This is equivalent to filling with mode, because the mode is 0.0 for both of those. The advantage of using`SimpleImputer` is speed and (in this case) consistency.

[43]:
```
cleaning.info(df).head(6).style.bar(subset=["nan"])
```

[43]: <pandas.io.formats.style.Styler at 0x1a75a241a60>

[44]:
```
classifier_pipe = Pipeline([
    ("cat_encoder", DummyEncoder(drop_first=True)),
    ("corr_trimmer", SmartCorrelatedSelection(selection_method="variance")),
    ("imputer", SimpleImputer(strategy="constant", fill_value=0.0)),
    ("classifier", DummyClassifier())
])

classifier_pipe
```

[44]:
```
Pipeline(steps=[('cat_encoder', DummyEncoder(drop_first=True)),
                ('corr_trimmer',
                 SmartCorrelatedSelection(selection_method='variance')),
                ('imputer', SimpleImputer(fill_value=0.0, strategy='constant')),
                ('classifier', DummyClassifier())])
```

6.1.2 Dummy Model

I create a dummy model which always predicts the most frequent class (0.0, in this case). It's good to ensure that my models are better than an extremely dumb alternative. If the dummy model is good at all, it's due to pure luck.

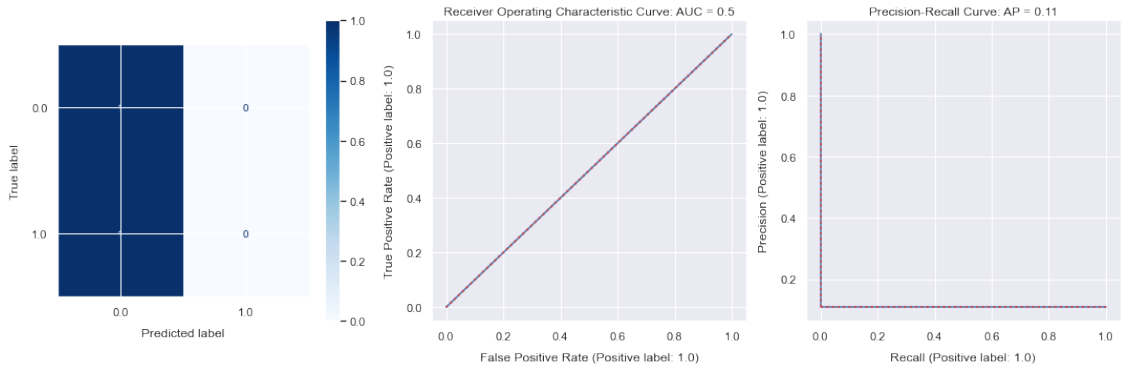
I train the model on all features except the target.

The confusion matrix indicates that the dummy gets 100% of the true negatives and 0% of the true positives, which is pretty bad.

The ROC curve is not even a curve, because if falls directly on the 1:1 line, with 0.5 AUC. The Precision-Recall Curve is a right angle. The balanced accuracy score is exactly 0.5.

[45]:
```
classifier_pipe["classifier"].set_params(strategy="most_frequent")
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test, zero_division=0)
```

<pandas.io.formats.style.Styler at 0x1a758f12310>



6.1.3 Baseline Logistic Regression

Since I haven't scaled the data yet, I set the solver to "lbfgs". The faster solver, "saga", is only fast on scaled data.

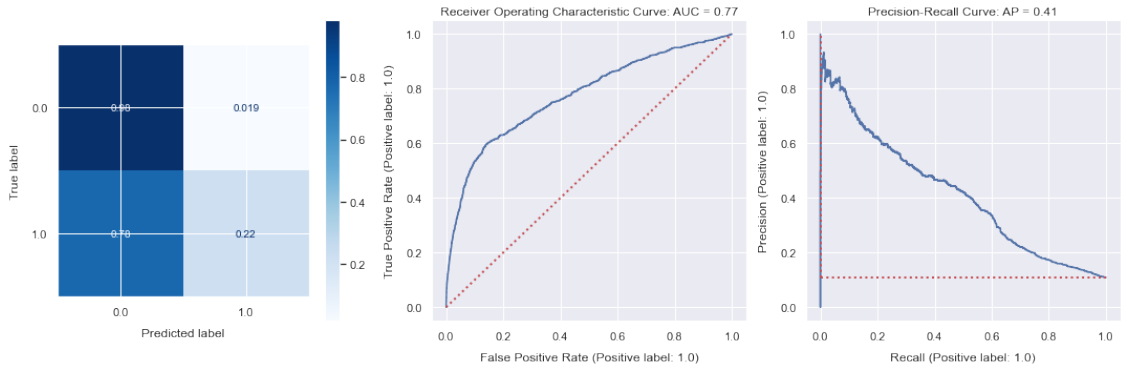
[46]:
```
logit = LogisticRegression(fit_intercept=False,
                           penalty="none",
                           multi_class="ovr",
                           max_iter=1e4,
                           warm_start=False,
                           solver="lbfgs")
logit
```

[46]:
```
LogisticRegression(fit_intercept=False, max_iter=10000.0, multi_class='ovr',
                   penalty='none')
```

It's better than the dummy, but not that much. Notice that the predictions are very biased towards the negative. The positive recall is terrible, and that will be the most important thing to fix. Ultimately, I want to see a confusion matrix with a strong diagonal. Going forward I will prioritize Average Precision (related to the area under the Precision-Recall Curve on the right) over AUC, as it is a better metric for finding balance with imbalanced data.

[47]:
```
classifier_pipe.set_params(classifier=logit)
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

```
<pandas.io.formats.style.Styler at 0x1a750bfaf10>
```



These are the features that were dropped because of multi-collinearity concerns. The SmartCorrelatedSelection device kept "n_employed", "prev_contact", and "n_employed". Based on what I know from the earlier plot of correlations, these are indeed smart choices, as they have the strongest effect on the target.

[48]:
```
print("correlated feature sets:")
display(classifier_pipe["corr_trimmer"].correlated_feature_sets_)
print("dropped:")
classifier_pipe["corr_trimmer"].features_to_drop_
```

```
correlated feature sets:
```

```
[{'emp_var_rate', 'euribor_3m', 'n_employed'},
 {'prev_contact', 'prev_failure'},
 {'prev_success', 'recent_prev_contact'}]
```

```
dropped:
```

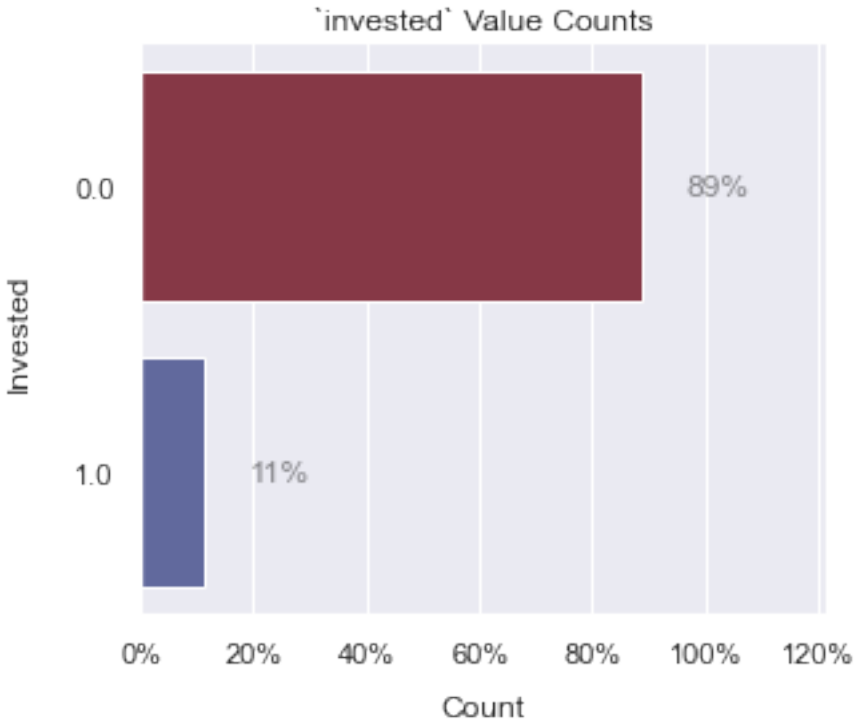
[48]:
```
['emp_var_rate', 'euribor_3m', 'prev_failure', 'recent_prev_contact']
```

6.2 Second Model

6.2.1 Balance Class Weight

My classes are very imbalanced with an almost 9:1 ratio. Fortunately the `LogisticRegression` estimator has a setting to automatically assign the classes balanced weights that are inversely proportional to their prevalence or "support". This is probably the single most important change that needs to be made to improve the model.

[49]:
```
fig = plotting.multi_countplot(data=y.to_frame(),
                               normalize=True,
                               orient="h",
                               height=4)
```



[50]:
```
logit.set_params(class_weight="balanced")
classifier_pipe
```

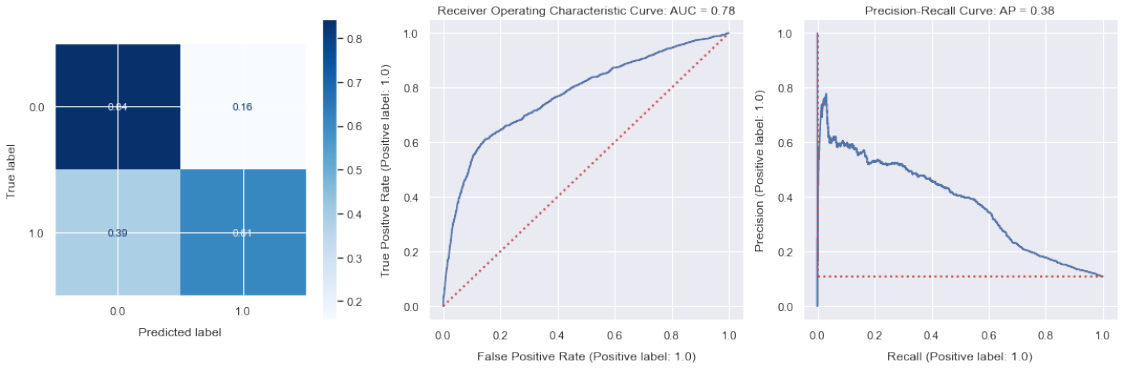
[50]:
```
Pipeline(steps=[('cat_encoder', DummyEncoder(drop_first=True)),
                ('corr_trimmer',
                 SmartCorrelatedSelection(selection_method='variance',
                                          variables=['age', 'housing', 'loan',
                                                     'contact_count',
                                                     'emp_var_rate',
                                                     'cons_price_idx',
                                                     'cons_conf_idx',
                                                     'euribor_3m', 'n_employed',
                                                     'prev_failure',
                                                     'prev_success',
                                                     'recent_prev_contact',
                                                     'prev_contact',
                                                     'contact_cellular',
                                                     'job_blue…
                                                     'job_technician',
                                                     'job_unemployed',
                                                     'marital_married',
                                                     'marital_single',
                                                     'education_basic_6y',
                                                     'education_basic_9y',
                                                     'education_high_school',
 'education_professional_course', …])),
                ('imputer', SimpleImputer(fill_value=0.0, strategy='constant')),
                ('classifier',
                 LogisticRegression(class_weight='balanced',
                                    fit_intercept=False, max_iter=10000.0,
                                    multi_class='ovr', penalty='none'))])
```

6.2.2 Train and Test

The positive recall has gone from 0.2 to 0.6, a major improvement. Now the model is actually useful, since it might actually predict an investor.

[51]:
```
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

```
<pandas.io.formats.style.Styler at 0x1a75a5c6550>
```



6.3 Third Model

6.3.1 Standard Scaling

I scale the data with `StandardScaler`, which centers on the mean and scales to standard deviation. This works on both categorical and numeric variables, since categorical variables are encoded as binary floats early on. `RobustScaler` would be a reasonable alternative, but centering on the median would radically skew the distributions of binary-encoded categorical variables.

[52]:
```
classifier_pipe = Pipeline([
    ("cat_encoder", DummyEncoder(drop_first=True)),
    ("corr_trimmer", SmartCorrelatedSelection(selection_method="variance")),
    ("scaler", StandardScaler()),
    ("imputer", SimpleImputer()),
    ("classifier", logit),
])
logit.set_params(solver="saga", penalty="none", C=1, max_iter=1e3)
classifier_pipe
```

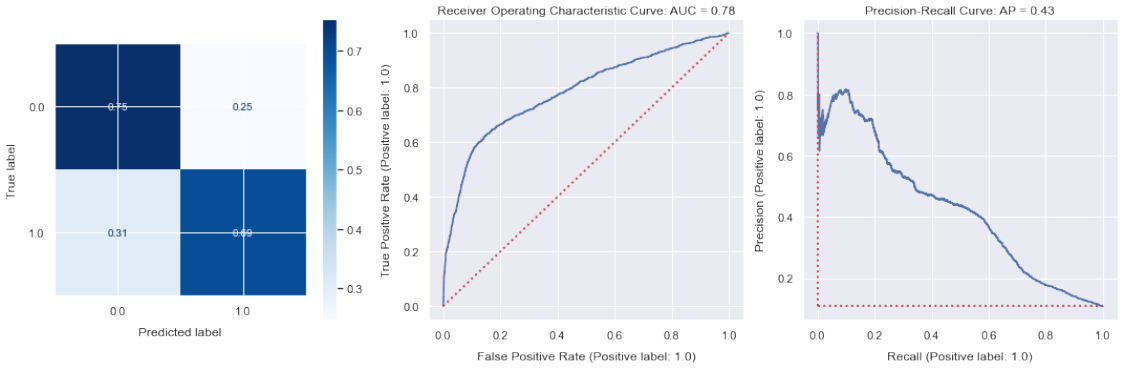
[52]:
```
Pipeline(steps=[('cat_encoder', DummyEncoder(drop_first=True)),
                ('corr_trimmer',
                 SmartCorrelatedSelection(selection_method='variance')),
                ('scaler', StandardScaler()), ('imputer', SimpleImputer()),
                ('classifier',
                 LogisticRegression(C=1, class_weight='balanced',
                                    fit_intercept=False, max_iter=1000.0,
                                    multi_class='ovr', penalty='none',
                                    solver='saga'))])
```

6.3.2 Train and Test

It's a major improvement. Positive recall is just under 0.7 now. The AP score has gone up significantly as well.

[53]:
```
logit.set_params(warm_start=False)
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

```
<pandas.io.formats.style.Styler at 0x1a75a648640>
```



6.4 Fourth Model

6.4.1 Winsorize before Scaling

Since my distributions are variously skewed and non-normal, I Winsorize them at 95% in order to reduce the influence of outliers on each distribution's mean and make them more suitable for `StandardScaler`.

[54]:
```
winsorizer = Winsorizer(variables=utils.true_numeric_cols(X),
                        capping_method="quantiles",
                        tail="both",
                        fold=0.05)
```

[55]:
```
classifier_pipe = Pipeline([
    ("cat_encoder", DummyEncoder(drop_first=True)),
    ("winsorizer", winsorizer),
    ("corr_trimmer", SmartCorrelatedSelection(selection_method="variance")),
    ("scaler", StandardScaler()),
    ("imputer", SimpleImputer()),
    ("classifier", logit),
])
logit.set_params(penalty="none", C=1)
classifier_pipe
```

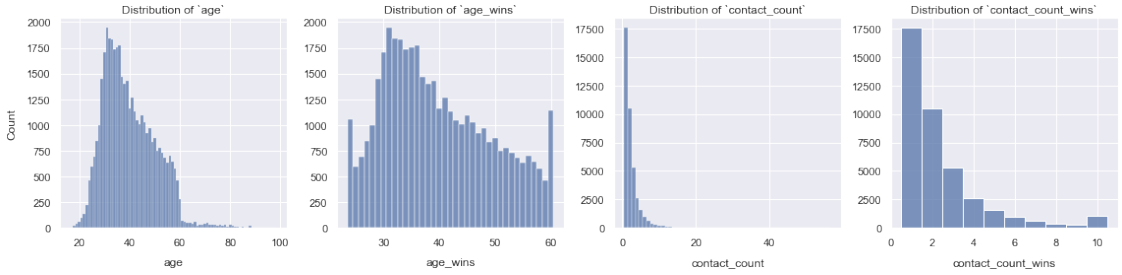
[55]:
```
Pipeline(steps=[('cat_encoder', DummyEncoder(drop_first=True)),
                ('winsorizer',
                 Winsorizer(capping_method='quantiles', fold=0.05, tail='both',
                            variables=['age', 'contact_count', 'emp_var_rate',
                                       'cons_price_idx', 'cons_conf_idx',
```

```
                                'euribor_3m', 'n_employed'])),
                ('corr_trimmer',
                 SmartCorrelatedSelection(selection_method='variance')),
                ('scaler', StandardScaler()), ('imputer', SimpleImputer()),
                ('classifier',
                 LogisticRegression(C=1, class_weight='balanced',
                                    fit_intercept=False, max_iter=1000.0,
                                    multi_class='ovr', penalty='none',
                                    solver='saga'))])
```

To Winsorize a distribution at 95% is to clip it to the outermost values between the 2.5th and 97.5th percentiles. As you can see from the examples below, this means moving far-out data points to new locations within the inner 95% range.

[56]:
```
examples = df[["age", "contact_count"]]
win_examples = outliers.quantile_winsorize(examples, inner=.95)
examples = examples.join(win_examples, rsuffix="_wins").sort_index(axis=1)

fig = plotting.multi_dist(data=examples, ncols=4, discrete=True, height=4);
fig.tight_layout()
del fig, examples, win_examples
```

```
              n_winsorized  pct_winsorized
age                   1461        3.565502
contact_count          869        2.120754
total_obs             2318        5.656970
```

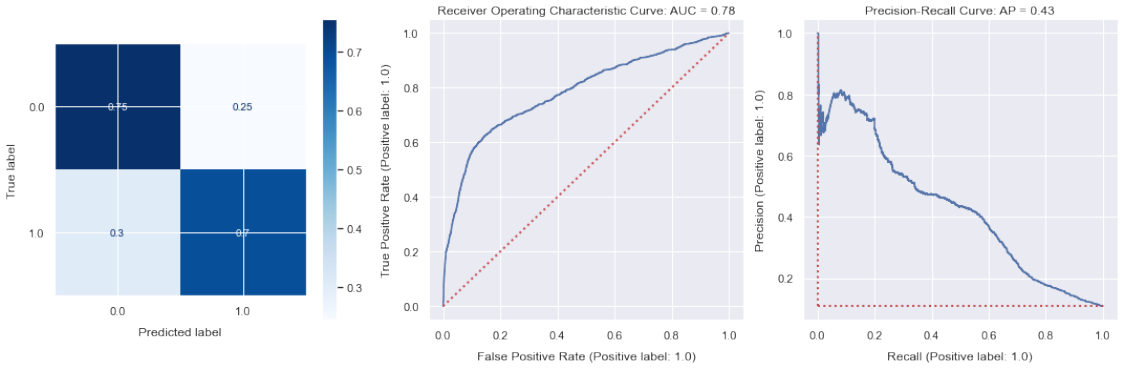


6.4.2 Train and Test

The Winsorization slightly increases the recall, weighted f1-score, and balanced accuracy, but doesn't make a huge difference overall. At the very least, it makes me more comfortable using `StandardScalar`, because outliers and skewness will have less of an effect on the means.

[57]:
```
classifier_pipe["winsorizer"].set_params(fold=0.05)
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

```
<pandas.io.formats.style.Styler at 0x1a750a1cfa0>
```



6.5 Final Model

6.5.1 Hyperparameter Tuning

It would be wise to tune the Winsorization setting alongside the (inverse) regularization strength to find the perfect balance. I set up a parameter grid and run a grid search on `classifier_pipe`.

[58]:
```
grid = dict(classifier__C=np.geomspace(1e-5, 1e5, 11),
            winsorizer__fold=np.linspace(.0, .2, 5))
grid
```

[58]:
```
{'classifier__C': array([1.e-05, 1.e-04, 1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01,
1.e+02,
        1.e+03, 1.e+04, 1.e+05]),
 'winsorizer__fold': array([0.  , 0.05, 0.1 , 0.15, 0.2 ])}
```

[76]:
```
# Cross validator which repeats to ensure accuracy
validator = RepeatedKFold(n_splits=5, n_repeats=10, random_state=64)

# Metrics to use
scoring = ["average_precision",
           "roc_auc_ovr_weighted",
           "balanced_accuracy"]

# Grid searcher
search = GridSearchCV(classifier_pipe,
                      grid,
                      n_jobs=-1,
                      cv=validator,
                      scoring=scoring,
                      refit=False)

logit.set_params(penalty="l2")
search
```

[76]:
```
GridSearchCV(cv=RepeatedKFold(n_repeats=10, n_splits=5, random_state=64),
             estimator=Pipeline(steps=[('cat_encoder',
                                        DummyEncoder(drop_first=True)),
                                       ('winsorizer',
                                        Winsorizer(capping_method='quantiles',
                                                   fold=0.05, tail='both',
                                                   variables=['age',
                                                              'contact_count',
                                                              'emp_var_rate',
                                                              'cons_price_idx',
                                                              'cons_conf_idx',
                                                              'euribor_3m',
                                                              'n_employed'])),
                                       ('corr_trimmer',
                                        SmartCo…
class_weight='balanced',
                                                                fit_intercept=False,
                                                                max_iter=1000.0,
                                                                multi_class='ovr',
                                                                solver='saga'))]),
             n_jobs=-1,
             param_grid={'classifier__C': array([1.e-05, 1.e-04, 1.e-03, 1.e-02,
1.e-01, 1.e+00, 1.e+01, 1.e+02,
       1.e+03, 1.e+04, 1.e+05]),
                         'winsorizer__fold': array([0.  , 0.05, 0.1 , 0.15, 0.2
])},
             refit=False,
             scoring=['average_precision', 'roc_auc_ovr_weighted',
                      'balanced_accuracy'])
```

Here I run the actual search. This cell should be kept frozen using the "Freeze" notebook extension, because it takes some time.

[63]:
```
# logit.set_params(warm_start=True)
# search.fit(X, y)
# results = pd.DataFrame(search.cv_results_)
# results.to_csv(normpath("sweep_results/logit_C_wins_results.csv"))
# results.head()
```

[63]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	\
0	1.146179	0.122440	0.19794	0.062599	
1	1.118600	0.094896	0.18664	0.050870	
2	1.069319	0.058387	0.16952	0.025904	
3	1.056280	0.061087	0.16514	0.018368	
4	1.022999	0.051021	0.16126	0.013645	

	param_classifier__C	param_winsorizer__fold	\
0	0.00001	0.0	

```
1            0.00001                   0.05
2            0.00001                    0.1
3            0.00001                   0.15
4            0.00001                    0.2

                                          params  \
0  {'classifier__C': 1e-05, 'winsorizer__fold': 0.0}
1  {'classifier__C': 1e-05, 'winsorizer__fold': 0…
2  {'classifier__C': 1e-05, 'winsorizer__fold': 0.1}
3  {'classifier__C': 1e-05, 'winsorizer__fold': 0…
4  {'classifier__C': 1e-05, 'winsorizer__fold': 0.2}

   split0_test_average_precision  split1_test_average_precision  \
0                       0.377076                       0.415353
1                       0.375898                       0.412781
2                       0.365647                       0.399019
3                       0.359139                       0.390178
4                       0.358637                       0.389577

   split2_test_average_precision  …  split43_test_balanced_accuracy  \
0                       0.396761  …                        0.703758
1                       0.396838  …                        0.705615
2                       0.384701  …                        0.703975
3                       0.377180  …                        0.701922
4                       0.376586  …                        0.702885

   split44_test_balanced_accuracy  split45_test_balanced_accuracy  \
0                        0.699363                        0.700959
1                        0.698470                        0.701167
2                        0.695044                        0.699659
3                        0.691478                        0.698014
4                        0.691754                        0.697602

   split46_test_balanced_accuracy  split47_test_balanced_accuracy  \
0                        0.699436                        0.719713
1                        0.701364                        0.718344
2                        0.698347                        0.716429
3                        0.697048                        0.713961
4                        0.697257                        0.713689

   split48_test_balanced_accuracy  split49_test_balanced_accuracy  \
0                        0.713453                        0.690281
1                        0.712014                        0.686576
2                        0.708722                        0.683836
3                        0.706805                        0.680954
4                        0.705707                        0.680540
```

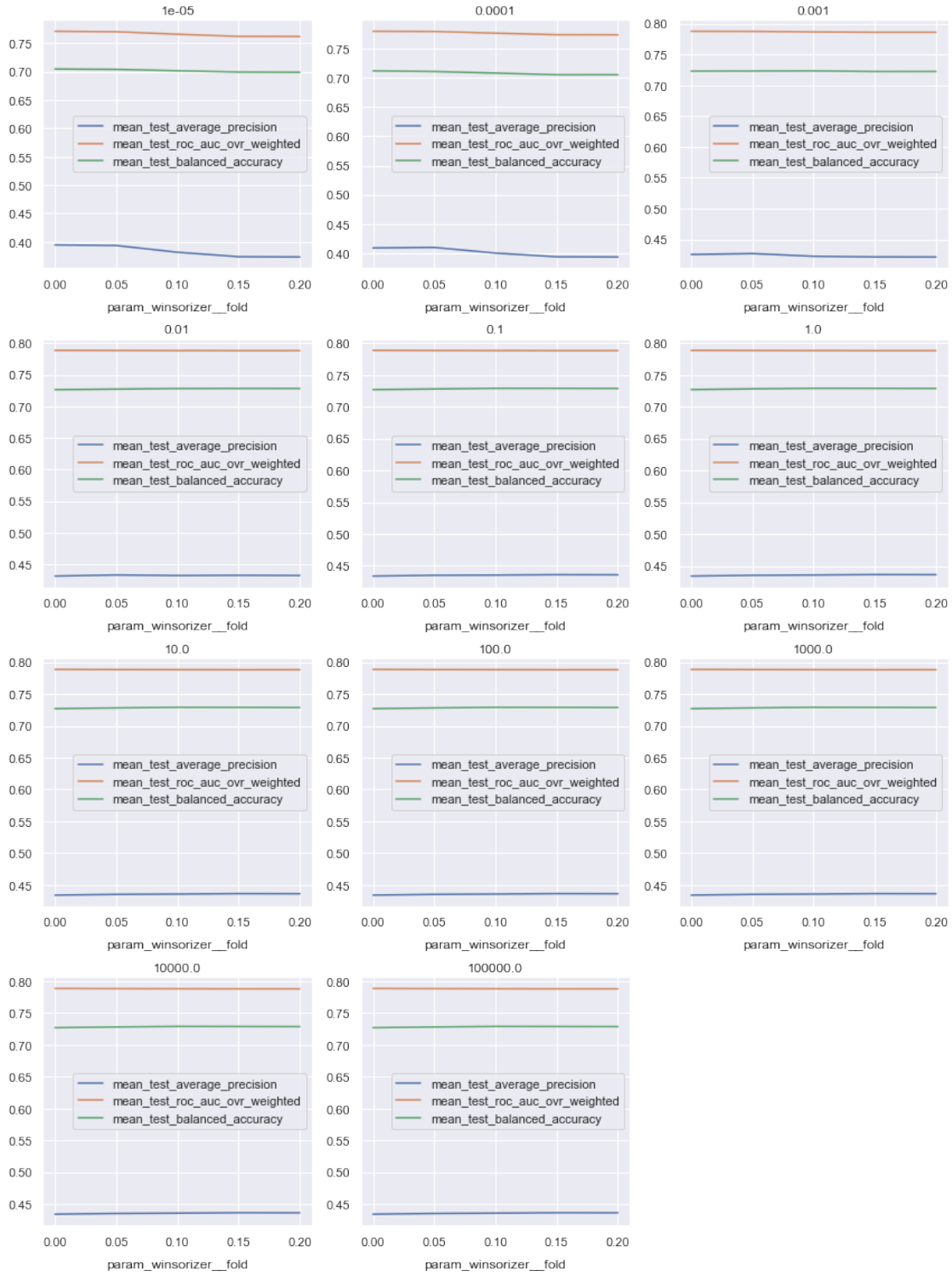
```
   mean_test_balanced_accuracy  std_test_balanced_accuracy  \
0                     0.704722                    0.007705
1                     0.704040                    0.007950
2                     0.701686                    0.007887
3                     0.699366                    0.008030
4                     0.699027                    0.007914

   rank_test_balanced_accuracy
0                           51
1                           52
2                           53
3                           54
4                           55

[5 rows x 166 columns]
```

[60]:
```
# Load the grid search results
results = pd.read_csv(
    normpath("sweep_results/logit_C_wins_results.csv"), index_col=0)
results = selection.tidy_results(results)

# Sort the results and cut out the extra stuff
results.sort_values("param_winsorizer__fold", ascending=True, inplace=True)
scores = [f"mean_test_{x}" for x in scoring]
params = ["param_classifier__C", "param_winsorizer__fold"]
results = results[params].join(results.loc[:, scores])

results.head(5)
```

[60]:
```
    param_classifier__C  param_winsorizer__fold  mean_test_average_precision  \
0               0.00001                     0.0                     0.394610
30             10.00000                     0.0                     0.434002
50         100000.00000                     0.0                     0.434003
5               0.00010                     0.0                     0.408939
20              0.10000                     0.0                     0.433878

    mean_test_roc_auc_ovr_weighted  mean_test_balanced_accuracy
0                         0.771230                     0.704722
30                        0.788610                     0.726996
50                        0.788610                     0.726980
5                         0.779641                     0.711947
20                        0.788602                     0.726857
```

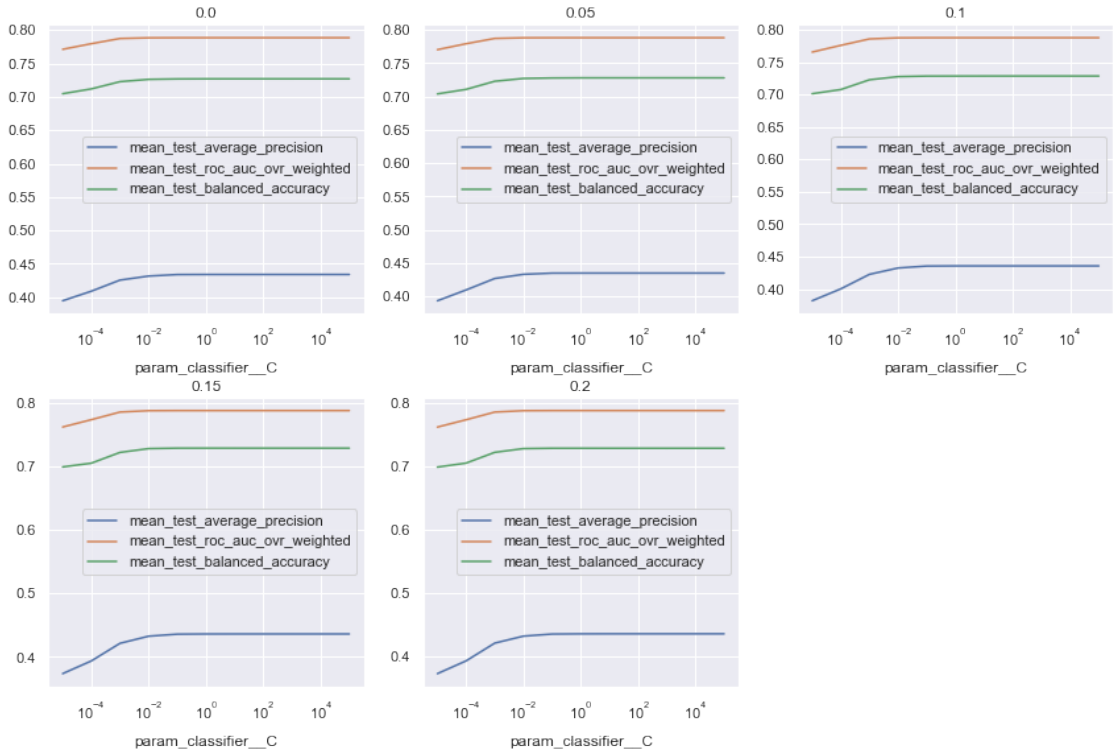
Winsorization Scores at Different C-values With high regularization, mean average precision peaks above fold=0.05 Winsorization. Mostly these are flat lines.

[61]:
```
plotting.grouper_plot(data=results,
                      grouper="param_classifier__C",
                      x="param_winsorizer__fold",
                      y=scores);
```



C-value Scores at Different Winsorization Thresholds Looks like strong regularization doesn't improve the model, and the scores plateau over 10^{-3}.

[62]:
```
fig = plotting.grouper_plot(data=results,
                            x="param_classifier__C",
                            grouper="param_winsorizer__fold",
                            y=scores);
for ax in fig.get_axes():
    ax.set_xscale("log")
```

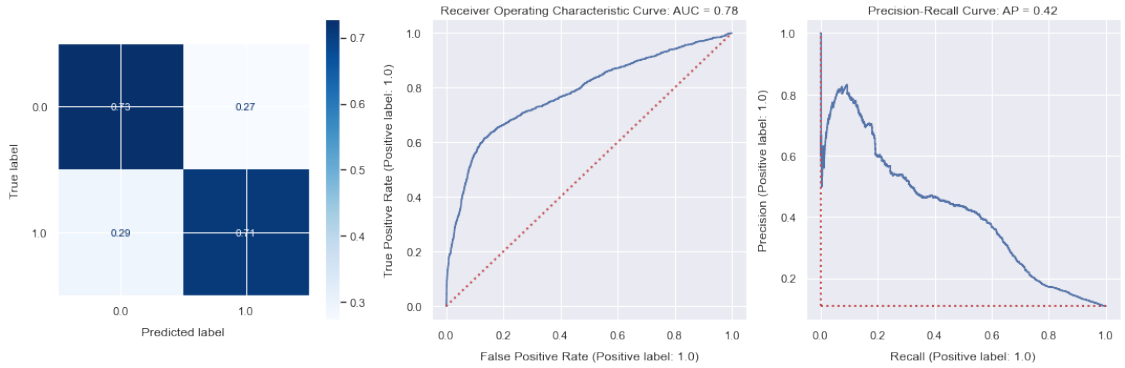


6.5.2 Train and Test

The grid search results indicate that average precision drops when C (inverse of regularization strength) is under 10^{-3} and that 0.05 is the best Winsorization setting. I set the regularization to 10^{-3} because it slightly increases positive recall without lowering the other scores too much. Look at that diagonal!

[63]:
```
logit.set_params(warm_start=False, penalty="l2", C=1e-3)
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

```
<pandas.io.formats.style.Styler at 0x1a755d681c0>
```



6.5.3 Retrain

Now that I've landed on a final set of preprocessors and parameters, I retrain the model on the whole dataset and save it.

[64]:
```
classifier_pipe.fit(X, y)
joblib.dump(classifier_pipe, normpath("models/final_model.joblib"))
```

[64]:
```
['models\\final_model.joblib']
```

7 Interpretation

Now to see which features are most important for prediction. These features and the strengths of their relationships with the target variable will be crucial information for marketing managers at Banco de Portugal.

First I gather the feature names from my `DummyEncoder` and subtract the ones dropped due to collinearity.

[65]:
```
feat_names = classifier_pipe["cat_encoder"].feature_names_.tolist()
for name in classifier_pipe["corr_trimmer"].features_to_drop_:
    print(f"'{name}' was dropped")
    feat_names.remove(name)
feat_names
```

```
'emp_var_rate' was dropped
'euribor_3m' was dropped
'prev_failure' was dropped
'recent_prev_contact' was dropped
```

[65]:
```
['age',
 'housing',
 'loan',
```

```
 'contact_count',
 'cons_price_idx',
 'cons_conf_idx',
 'n_employed',
 'prev_success',
 'prev_contact',
 'contact_cellular',
 'job_blue_collar',
 'job_entrepreneur',
 'job_housemaid',
 'job_management',
 'job_retired',
 'job_self_employed',
 'job_services',
 'job_student',
 'job_technician',
 'job_unemployed',
 'marital_married',
 'marital_single',
 'education_basic_6y',
 'education_basic_9y',
 'education_high_school',
 'education_professional_course',
 'education_university_degree',
 'contact_month_apr',
 'contact_month_may',
 'contact_month_jun',
 'contact_month_jul',
 'contact_month_aug',
 'contact_month_sep',
 'contact_month_oct',
 'contact_month_nov',
 'contact_weekday_tue',
 'contact_weekday_wed',
 'contact_weekday_thu',
 'contact_weekday_fri']
```

Then I grab the coefficients and put them alongside their names.

[66]:
```
coef = pd.DataFrame(logit.coef_.T,
                    columns=["coef"],
                    index=feat_names).squeeze().copy()
coef.sort_values().head()
```

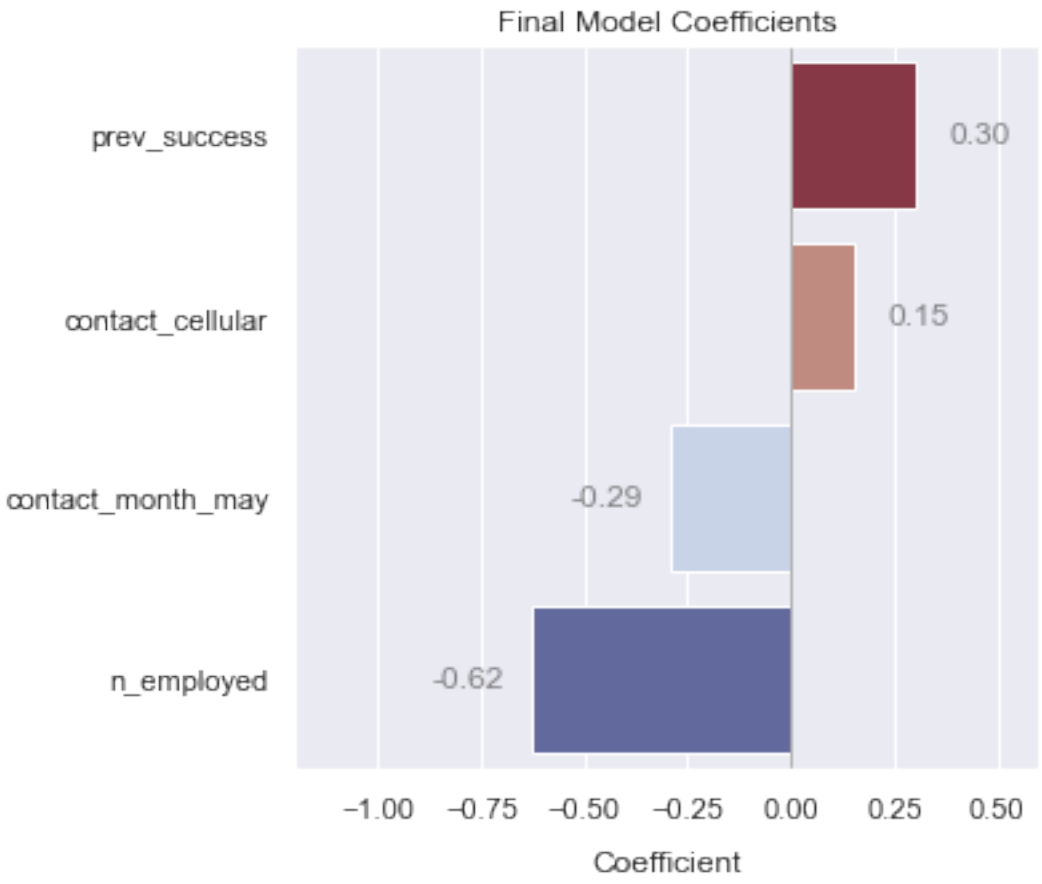
[66]:
```
n_employed          -0.624929
contact_month_may   -0.288588
contact_month_nov   -0.090772
prev_contact        -0.081283
```

```
contact_month_aug   -0.064635
Name: coef, dtype: float64
```

Then I slice out the larger ones and plot them.

[67]:
```
# Slice out only the large coefficients;
# L2 regularization crushed most of them
major_coef = coef.loc[coef.abs() >= .1].copy()

# Plot heated bars
ax = plotting.heated_barplot(data=major_coef,
                             figsize=(5, 5))
plotting.annot_bars(ax, dist=.25)
ax.set_xlim(-1.2, .6)
ax.set(title="Final Model Coefficients", xlabel="Coefficient",)
```

[67]:
```
[Text(0.5, 1.0, 'Final Model Coefficients'), Text(0.5, 0, 'Coefficient')]
```



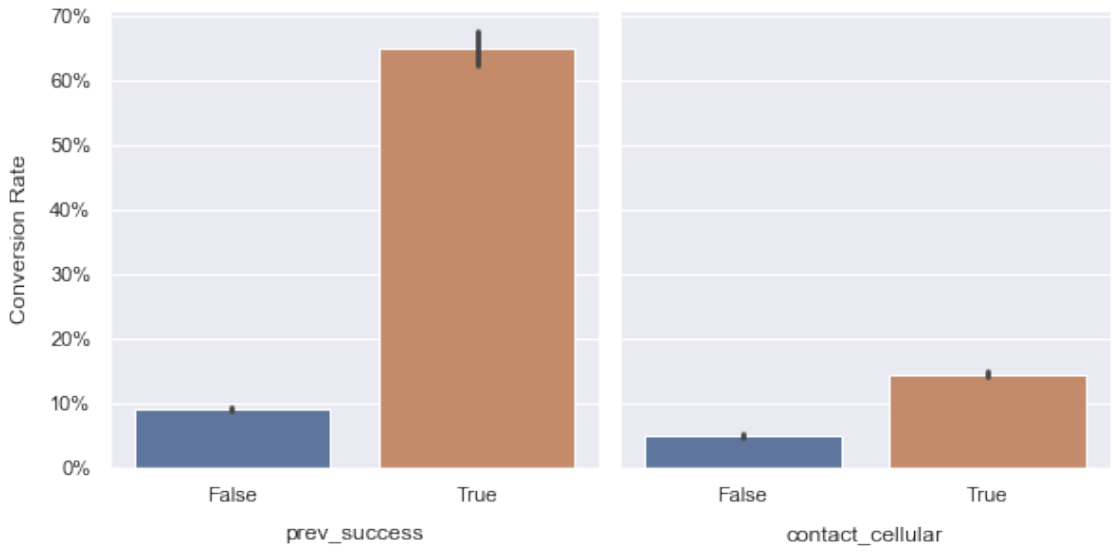
7.1 Positive Coefficients

The largest positive coefficient is 'prev_success', which is hardly surprising. Clients who previously invested in accounts because of the bank's marketing campaigns are likely to invest again. This is common sense, but it's good to see that the model aligns with common sense.

The second largest positive coefficient is 'contact_cellular', meaning that calling the client on a cell phone increases the probability of investment. Why? I don't know. Maybe people are more likely to take these calls on their cell phones than when they're at home with their families. Or perhaps it has to do with what kind of people used cell phones rather than landlines in 2008-2010. It's about half the magnitude of 'prev_success', indicating that it has a weaker relationship. That also comports with common sense.

[68]:
```
# Convert binary float variables to bool for plotting
df[utils.binary_cols(df)] = df.loc[:, utils.binary_cols].astype(np.bool_)

# Plot conversion rates for positive coeff features
fig = plotting.multi_rel(data=df,
                         x=["prev_success", "contact_cellular"],
                         y="invested",
                         kind="bar",
                         size=(4, 4))

# Format mean as percent
axs = fig.get_axes()
axs[0].yaxis.set_major_formatter(ticker.PercentFormatter(xmax=1))
axs[0].set(ylabel="Conversion Rate")
```

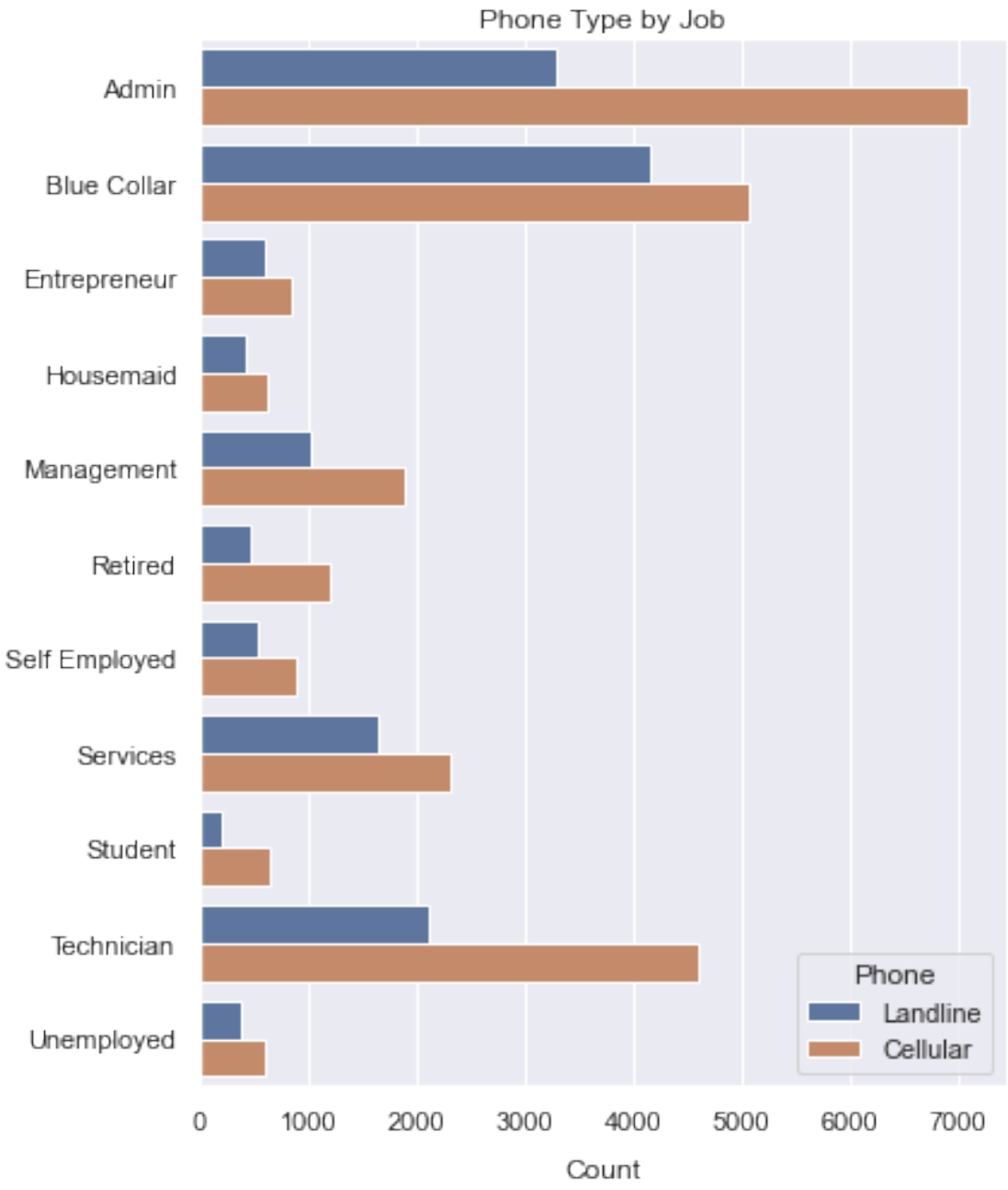
[68]:
```
[Text(73.125, 0.5, 'Conversion Rate')]
```



Well, even retirees were using cell phones apparently. I don't know why it's related to investing. Perhaps just because it's more popular?

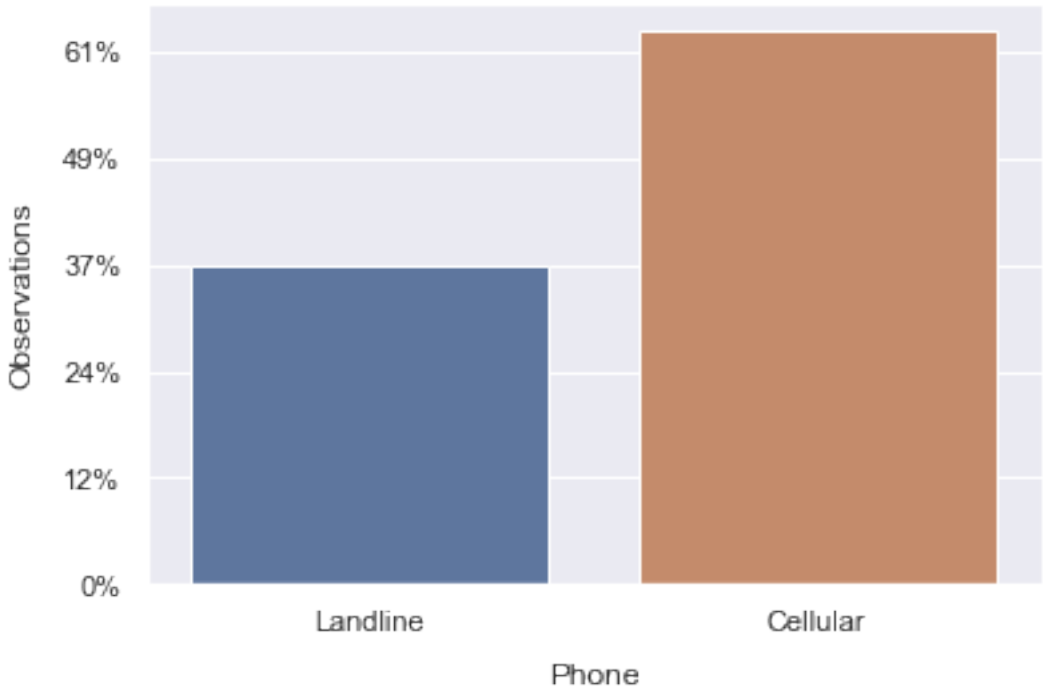
[69]:
```
# Create special title-formatted DataFrame
cell_df = utils.title_mode(df)
cell_df["Phone"] = cell_df["Contact Cellular"].map(
    lambda x: "Cellular" if x else "Landline")

# Make the graph
fig, ax = plt.subplots(figsize=(6, 8))
sns.countplot(data=cell_df, y="Job", hue="Phone", ax=ax)
ax.set(ylabel=None, xlabel="Count", title="Phone Type by Job")
# del cell_df
```

[69]:
```
[Text(0, 0.5, ''), Text(0.5, 0, 'Count'), Text(0.5, 1.0, 'Phone Type by Job')]
```



It is certainly more popular.

[70]:
```
ax = sns.countplot(data=cell_df, x="Phone")
ax.yaxis.set_major_formatter(ticker.PercentFormatter(xmax=cell_df.shape[0]))
ax.set(ylabel="Observations")
del cell_df
```



7.2 Negative Coefficients

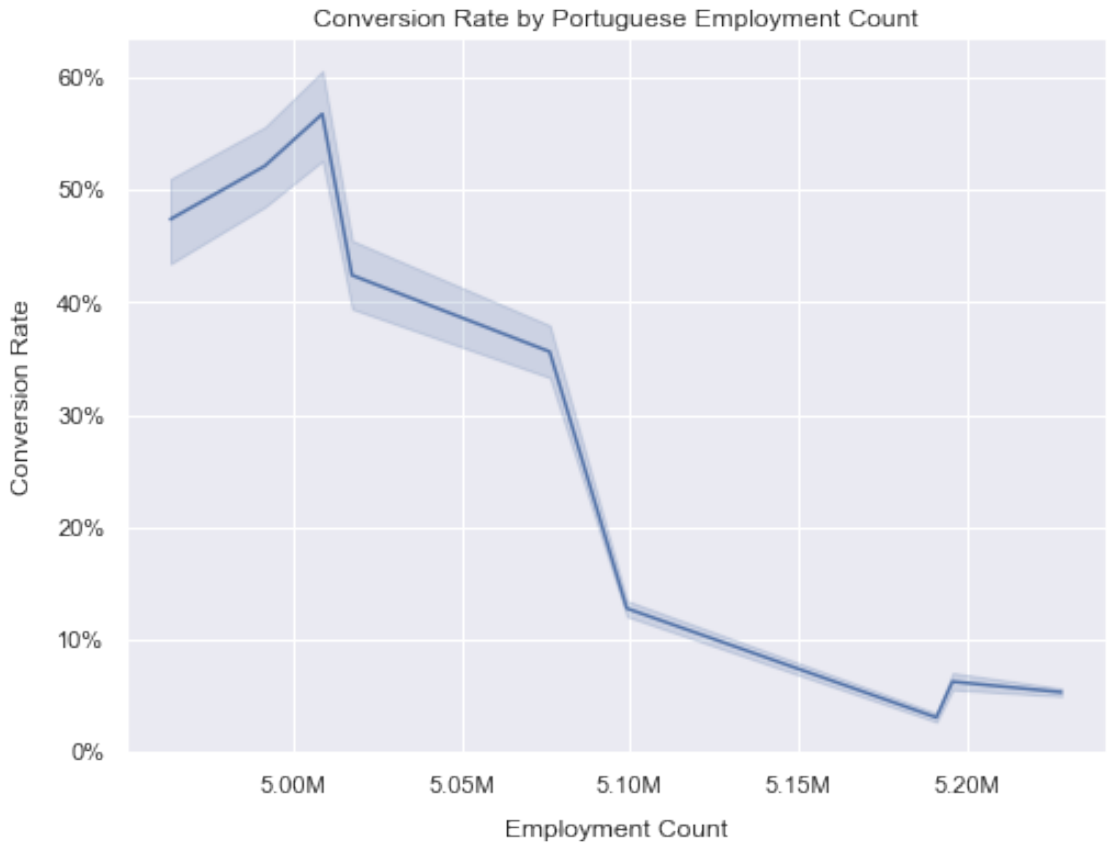
The largest negative coefficient (in magnitude)—and the largest coefficient of all by far—is 'n_employed'. This is a quarterly measure of how many people are employed in Portugal, in thousands. Its magnitude of 0.62 is more than twice that of 'prev_success' at 0.3. It's also over twice as important as 'contact_month_may', which is 0.29. Apparently May is just a terrible month for getting people to invest in term deposits. Or at least it was in 2008, 2009, and 2010.

As you can see from the graph below, the employment count has a clear negative relationship with the bank's conversion rate. What's unclear to me is why. Does employment decrease because of increased investment? That doesn't seem plausible, although I'm no economist. Do people invest while unemployed, or when the economy is slow? That doesn't seem plausible either.

[71]:
```
# Create special plotting DataFrame
neg_df = utils.title_mode(df)
neg_df["N Employed"] *= 1000

# Create Axes
fig, ax = plt.subplots(figsize=(8, 6))
sns.lineplot(data=neg_df, x="N Employed", y="Invested", ax=ax)

# Format and label
ax.xaxis.set_major_formatter(plotting.big_number_formatter(2))
ax.yaxis.set_major_formatter(ticker.PercentFormatter(xmax=1))
ax.set(ylabel="Conversion Rate",
```

```
        xlabel="Employment Count",
        title="Conversion Rate by Portuguese Employment Count")
```

[71]:
```
[Text(0, 0.5, 'Conversion Rate'),
 Text(0.5, 0, 'Employment Count'),
 Text(0.5, 1.0, 'Conversion Rate by Portuguese Employment Count')]
```



No, unemployed people don't invest a lot. And that makes sense, seeing as they probably don't have a lot of extra money laying around. It's surprising that they invest at all, actually. 14% conversion is not bad for people who don't have disposable income. Maybe these are more like Paris Hilton types.

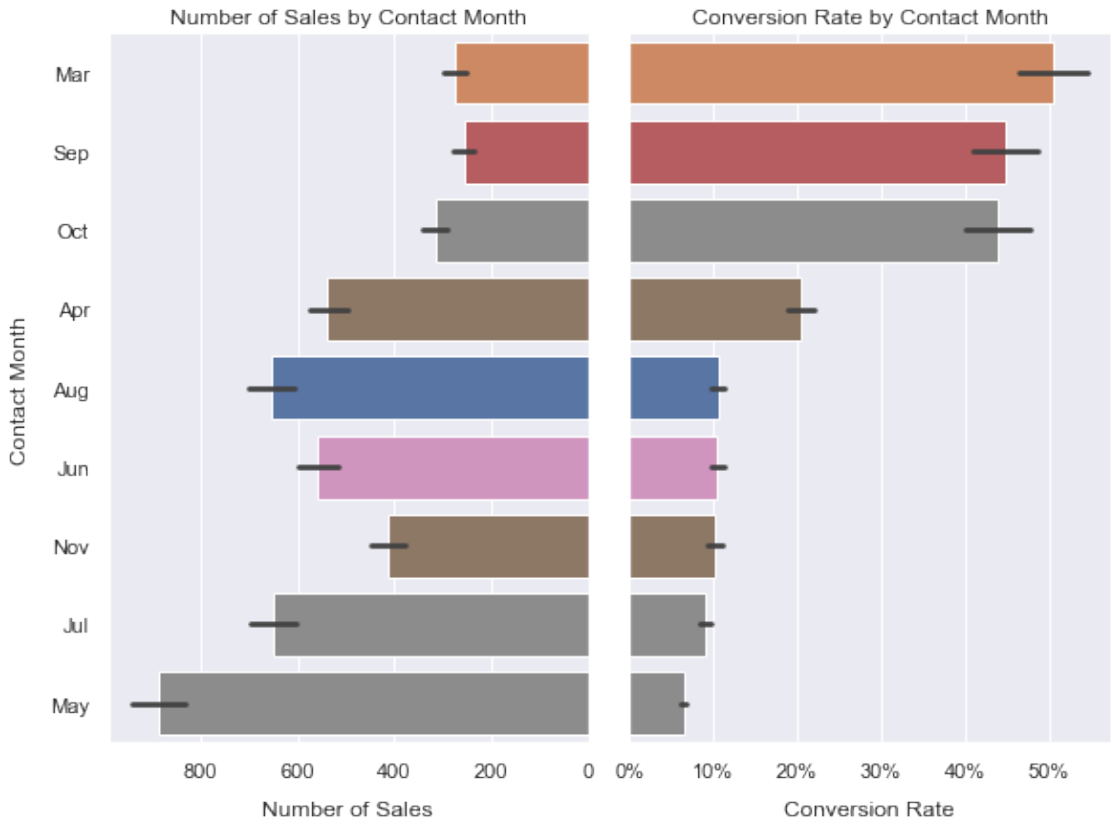
Anyway, regardless of *why* employment is inversely related to investment conversions, we can be certain that it is related in this way. The bank should invest more in marketing when employment is low.

[72]:
```
df.query("job == 'unemployed'")["invested"].value_counts(1)
```

[72]:
```
False    0.861554
True     0.138446
Name: invested, dtype: float64
```

The poor conversion for the month of May is similarly mysterious. Recall the figure below, reproduced from the exploration setting. Interestingly, May has the most total sales despite having the lowest conversion rate. This must mean that there are a LOT of observations in May.

[73]:
```
conversion_bars(data=df, y="contact_month", size=(4, 6));
```



Yep, for whatever reason, May has nearly 14,000 observations while the top 3 conversion rate months have observation counts in the hundreds.

Why are these categories so uneven, I wonder? Perhaps it has something to do with the fact that the top 4 highest observation count months are in the summer. Does the bank just spend more on marketing in the summer? Or is this just a very uneven sample of a larger dataset held by the bank? I don't know.

Anyway, the data clearly indicates that May is a very bad month for conversion efficiency.

[74]:
```
df["contact_month"].value_counts()
```

[74]:
```
may    13764
jul     7164
aug     6171
jun     5318
nov     4097
```

```
apr     2629
oct      717
sep      570
mar      546
Name: contact_month, dtype: int64
```

8 Recommendations

When employment is low, spend more resources on marketing the investment product. There is a very strong relationship between low employment and investment, although the true nature of the relationship remains a mystery. #### Do not spend resources on marketing in May. May is a strikingly bad month for conversion efficiency. It had the most total sales, but the worst ratio. #### Prioritize clients who have previously invested as a result of marketing efforts. This is probably already something you're doing, but rest assured that it works. #### Prioritize cellular clients. Clients who use a cell phone are more likely to invest than those who use a landline.

9 Future Work

The most important future work would be to build different types of models and compare them to my final `LogisticRegression`. `RandomForestClassifier`, `LinearSVC`, and `KNeighborsClassifier` are three obvious choices. Unlike most support vector machines, the `LinearSVC` is able to handle datasets with large numbers of observations. But as it is a linear model, I still have to worry about multicollinearity.

Multicollinearity is not a concern, however, with the `RandomForestClassifier` or the `KNeighborsClassifier`. That means no features have to be dropped on that account. This alone is reason to think one of these models could perform better than my regression. Of all of these, I see the most potential in the `RandomForestClassifier`, in part because it has so many hyperparameters to tune.

-