main_notebook

June 10, 2021

1 Predicting Bank Telemarketing Sales

- Nick Gigliotti
- ndgigliotti@gmail.com

Table of Contents

Predicting Bank Telemarketing Sales

Business Problem

Dataset

Feature Explanations

Initial Cleaning

Binary Categoricals

Multi-class Categoricals

Exploration

Modeling

Modeling Imports

Train-test Split

First Model

Baseline Preprocessors

Dummy Model

Baseline Logistic Regression

Second Model

Balance Class Weight

Train and Test

Third Model

Standard Scaling

Train and Test

Fourth Model

Winsorize before Scaling

Train and Test

Final Model

Hyperparameter Tuning

Train and Test

Retrain

Interpretation

Positive Coefficients

Negative Coefficients

Recommendations

Future Work

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
%load_ext nb_black
```

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

[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)
```

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

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	\
0	56	housemaid	married	basic.4y	no	no	no	telephone	
1	57	services	married	high.school	unknown	no	no	telephone	
2	37	services	married	high.school	no	yes	no	telephone	
3	40	admin.	married	basic.6y	no	no	no	telephone	
4	56	services	married	high.school	no	no	yes	telephone	

	month	day_of_week	…	campaign	pdays	previous	poutcome	emp.var.rate	\
0	may	mon	…	1	999	0	nonexistent	1.1	
1	may	mon	…	1	999	0	nonexistent	1.1	
2	may	mon	…	1	999	0	nonexistent	1.1	
3	may	mon	…	1	999	0	nonexistent	1.1	
4	may	mon	…	1	999	0	nonexistent	1.1	

	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y
0	93.994	-36.4	4.857	5191.0	no
1	93.994	-36.4	4.857	5191.0	no
2	93.994	-36.4	4.857	5191.0	no
3	93.994	-36.4	4.857	5191.0	no
4	93.994	-36.4	4.857	5191.0	no

```
[5 rows x 21 columns]
```

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

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
```

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

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
```

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

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]:
```
# Replace periods with underscores
df.columns = df.columns.str.replace(".", "_", regex=False)

# Map old names to new names
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"}

# Rename using dictionary
df.rename(columns=rename, inplace=True)

# Delete dictionary
del rename

# Display results
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')
```

```
<IPython.core.display.Javascript 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 checking the unique values for null placeholders. There are a few.

[7]:
```
cleaning.show_uniques(df, cut=20)
```

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

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

I replace the placeholders with NaN and survey the missing values and duplicates. There are 12 duplicate rows—an easy number to drop.

[8]:
```
df["days_since_prev"].replace(999, np.NaN, inplace=True)
df.replace(["unknown", "nonexistent"], np.NaN, inplace=True)
cleaning.info(df)
```

[8]:

	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

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

I drop the duplicate rows.

[9]:
```
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]
```

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

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

[10]:
```
cleaning.show_uniques(df, cut=20)
```

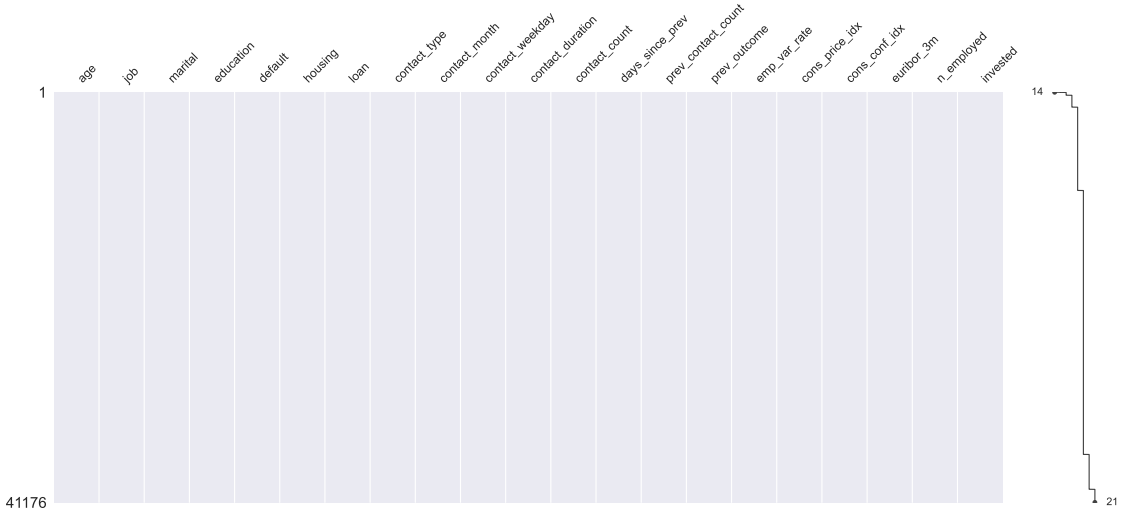
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript 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'.

[11]:
```
msno.matrix(df, sort="ascending")
```

[11]: <AxesSubplot:>




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

4.0.1 Binary Categoricals

I begin by encoding and engineering a number of binary categorical features.

A few columns, including the target, have 'yes' and 'no' strings as values. Since these are equivalent to 1 and 0, I go ahead and numerically encode them. I'd rather not have to one-hot encode them later and potentially turn each of them into two features.

I use the `strtobool` function from distutils (part of the Standard Library). It's a handy function which converts truth-value-indicating strings (e.g. 'y','n', 'yes', 'no', 't', 'f', 'true', 'false', 'on', 'off', '1', '0') to binary numeric values (1 and 0).

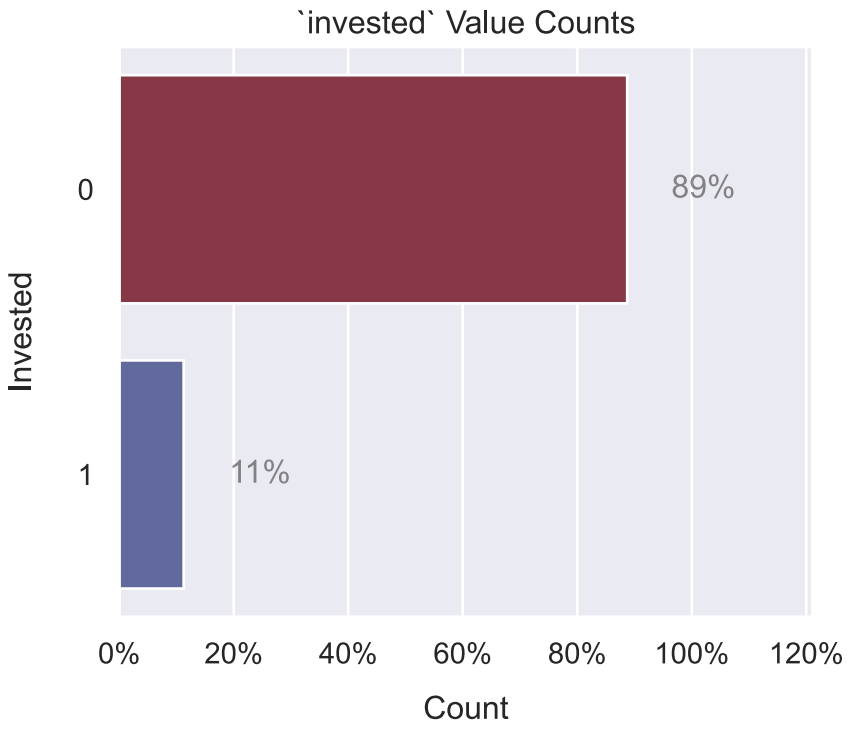
[12]:
```
# List columns with string 'yes' and 'no' values
string_cols = ["default", "housing", "loan", "invested"]

# Convert 'yes' and 'no' to 1 and 0
df[string_cols] = df[string_cols].applymap(strtobool, na_action="ignore")

# Display results
cleaning.show_uniques(df, columns=string_cols)

# Mop up the temporary variable
del string_cols
```

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

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

The classes of the prediction target, 'invested', are imbalanced at a ratio of roughly 9:1. This is a problem I'll have to deal with in the modeling phase.

[13]:
```
plotting.multi_countplot(data=df[["invested"]], normalize=True, height=4);
```



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

I one-hot encode 'prev_outcome', rendering nulls as 0 for both categories. 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.

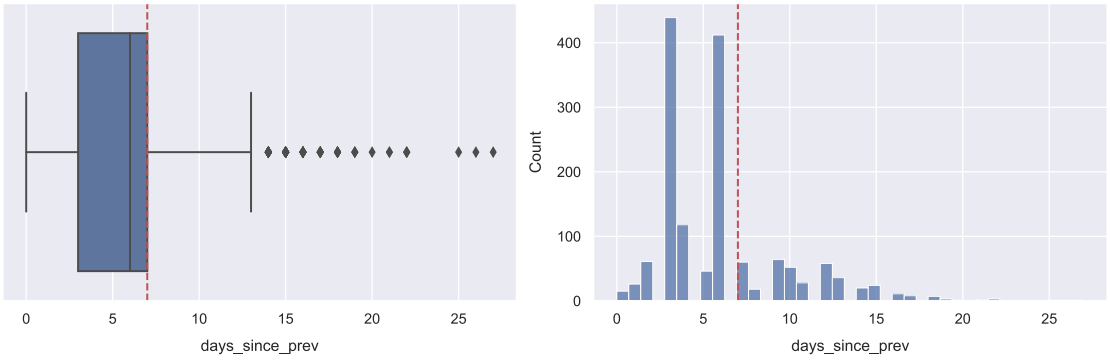
[14]:
```
# One-hot encode 'prev_outcome' categories: {'failure', 'success'}
df = pd.get_dummies(df, columns=["prev_outcome"], prefix="prev")

# Display value counts for the new features
df[["prev_failure", "prev_success"]].value_counts(normalize=True)
```

[14]:
```
prev_failure  prev_success
0             0              0.863391
1             0              0.103264
0             1              0.033345
dtype: float64
```

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

The third quartile of "days_since_prev" is exactly 7 days, as illustrated by the dashed red line on the boxplot below. This seems like a good cutoff for turning "days_since_prev" into a categorical.

[15]:
```
# 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 variables
del fig, ax1, ax2
```




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

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.

[16]:
```
# Create boolean feature
df["recent_prev_contact"] = df["days_since_prev"] <= 7

# Display distribution
df["recent_prev_contact"].value_counts(1)
```

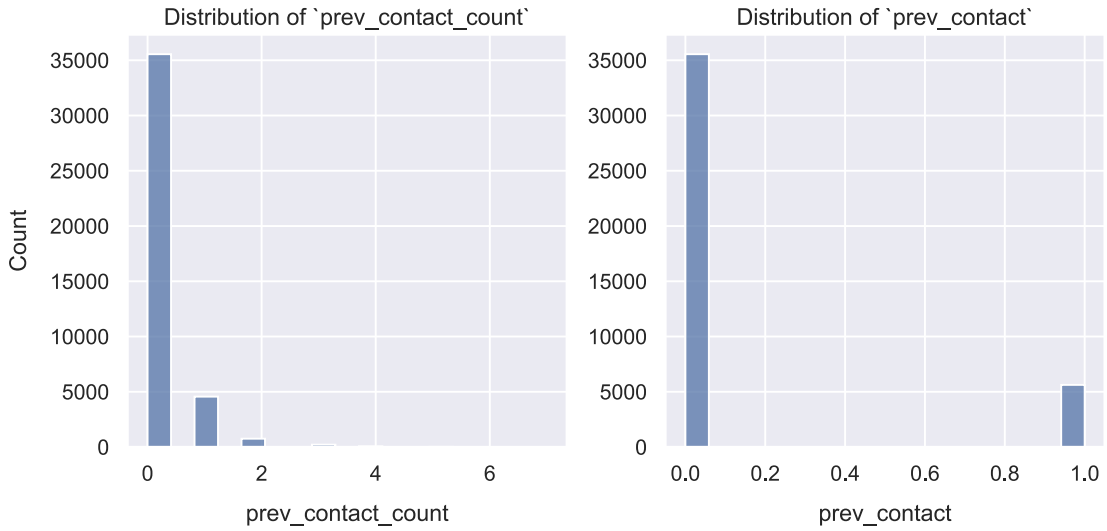
[16]:
```
False    0.971415
True     0.028585
Name: recent_prev_contact, dtype: float64
```

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

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.

[17]:
```
# 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"]], height=4);
```

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




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

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

[18]:
```
# Create boolean feature
df["contact_cellular"] = df["contact_type"] == "cellular"

# Display distribution
df["contact_cellular"].value_counts()
```

[18]:
```
True     26135
False    15041
Name: contact_cellular, dtype: int64
```

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

I drop the features which I've redesigned.

[19]:
```
# List features to drop
to_drop = ["days_since_prev", "prev_contact_count", "contact_type"]

# Drop them in place
df.drop(to_drop, axis=1, inplace=True)

# Show results
df.columns
```

[19]:
```
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')
```

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

I convert all binary categoricals to float, and avoid converting them to categorical dtype. It will be easier to work with them as numeric variables, since they don't need to be one-hot encoded.

[20]:
```
# Get list of binary features
binary_cats = utils.binary_cols(df)

# Convert to float
df[binary_cats] = df[binary_cats].astype(np.float64)

# Show results
cleaning.show_uniques(df, columns=binary_cats)
```

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

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

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

[21]:
```
cleaning.class_distrib(df[binary_cats], normalize=True)
```

[21]:

	min_members	max_members	classes
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

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

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.

[22]:
```
# Drop 'default' in place and remove from list
df.drop(columns=["default"], inplace=True)
binary_cats.remove("default")

# Show token and type counts
display(cleaning.class_distrib(df[binary_cats], normalize=True))

# Delete binary feature list
```

```
del binary_cats
```

```
                     min_members  max_members  classes
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
```

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

4.0.2 Multi-class Categoricals

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

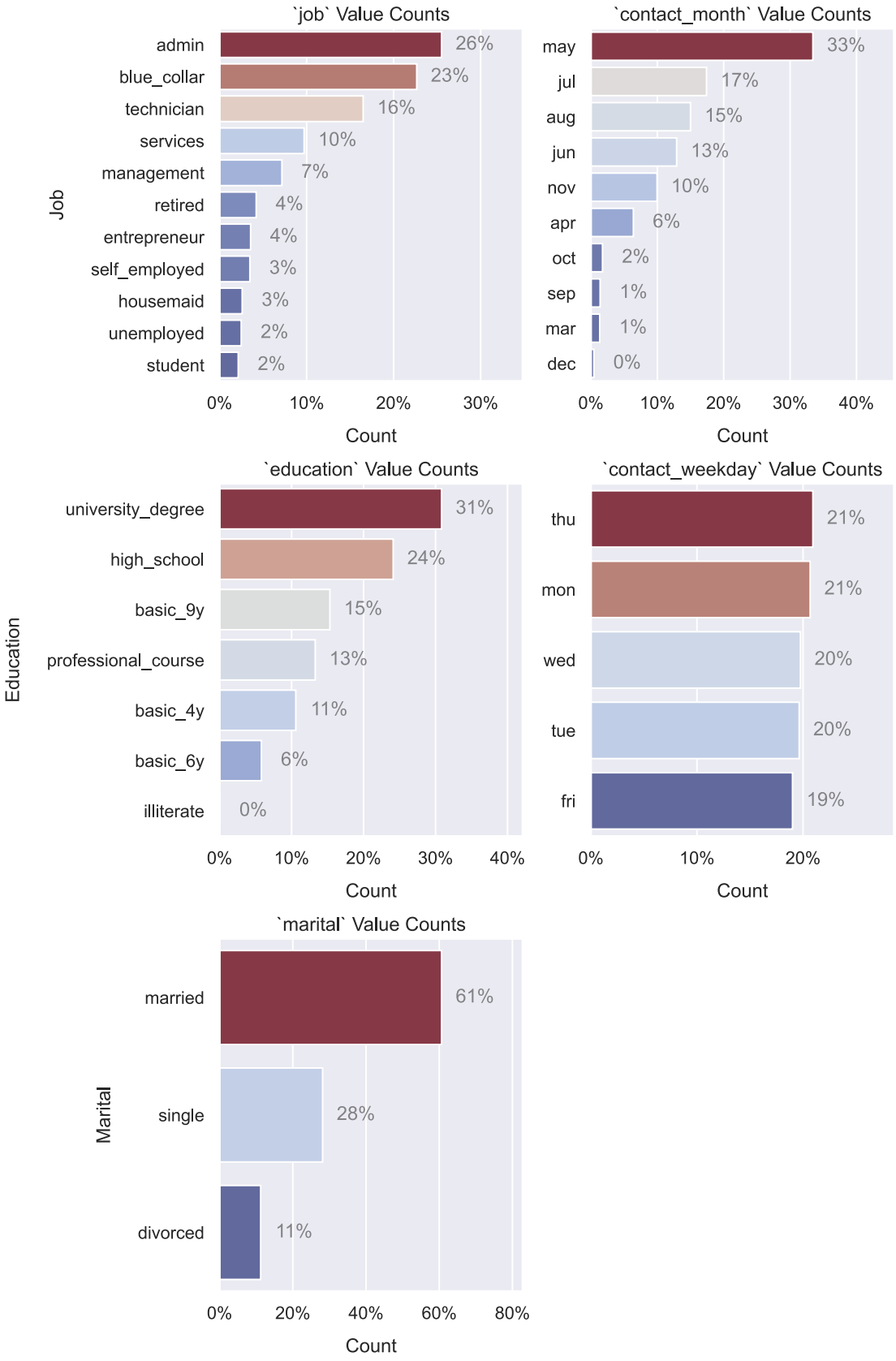
[23]:
```
# List multi-class categoricals
multi_cat = ["job", "marital", "education",
             "contact_month", "contact_weekday"]

# 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")

# Show results
cleaning.show_uniques(df, columns=multi_cat)
```

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

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

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

[24]:
```
# Plot distributions
plotting.multi_countplot(data=df[multi_cat],
                         normalize=True,
                         ncols=2,
                         height=4)

# Delete list of multi-class features
del multi_cat
```


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

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

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…

[25]:
```
# 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()

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

# Mop up variable
del keep
```

```
Dropped 200 observations.
```

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

I order the weekdays and months for plotting purposes.

[26]:
```
# 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 results
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')
```

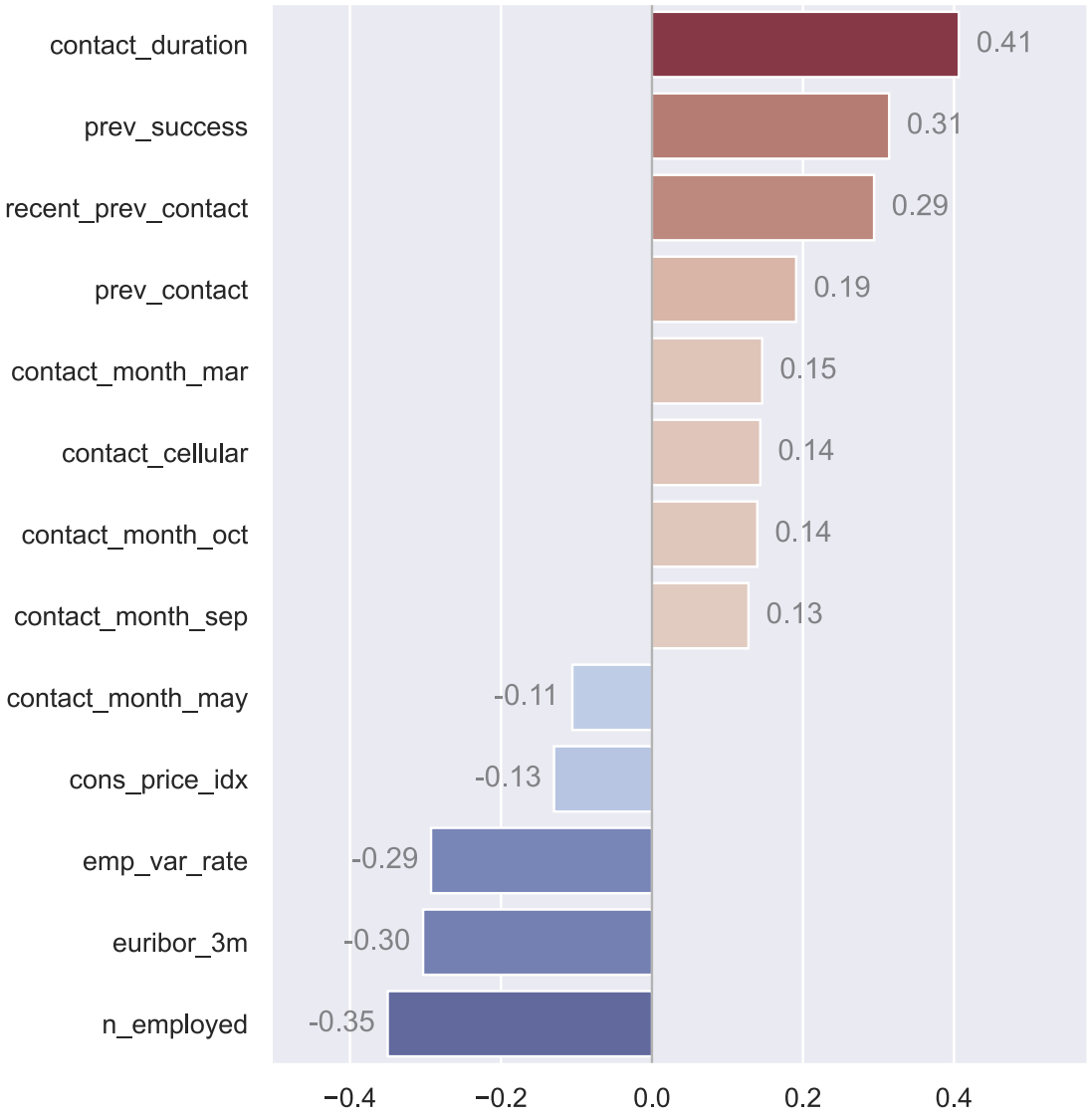
```
<IPython.core.display.Javascript object>
```

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

[27]:
```
corr_df = pd.get_dummies(df.drop(columns="invested"))
corr_df = corr_df.corrwith(df["invested"])
corr_df = corr_df.loc[corr_df.abs() > .1]
ax = plotting.heated_barplot(data=corr_df)
plotting.annot_bars(ax)
del corr_df
```

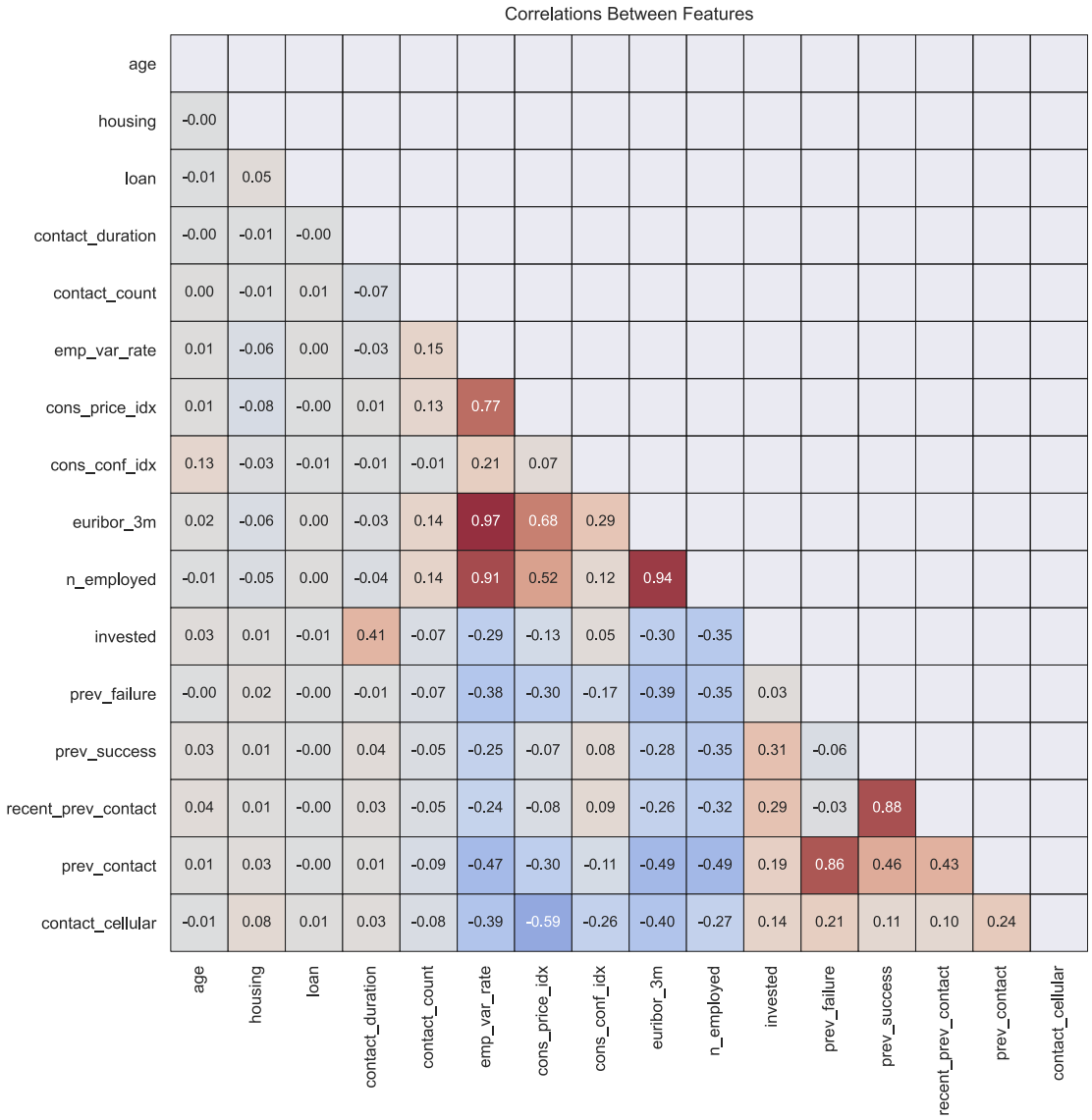



```
<IPython.core.display.Javascript object>
```

Some of the economic context variables like 'euribor_3m', 'n_employed' and 'emp_var_rate' are highly correlated. 'recent_prev_contact' is highly correlated with 'prev_success', as is 'rev_contact' with 'prev_failure'.

[28]:
```
plotting.pair_corr_heatmap(data=df, scale=.7)
```

[28]:
```
<AxesSubplot:title={'center':'Correlations Between Features'}>
```

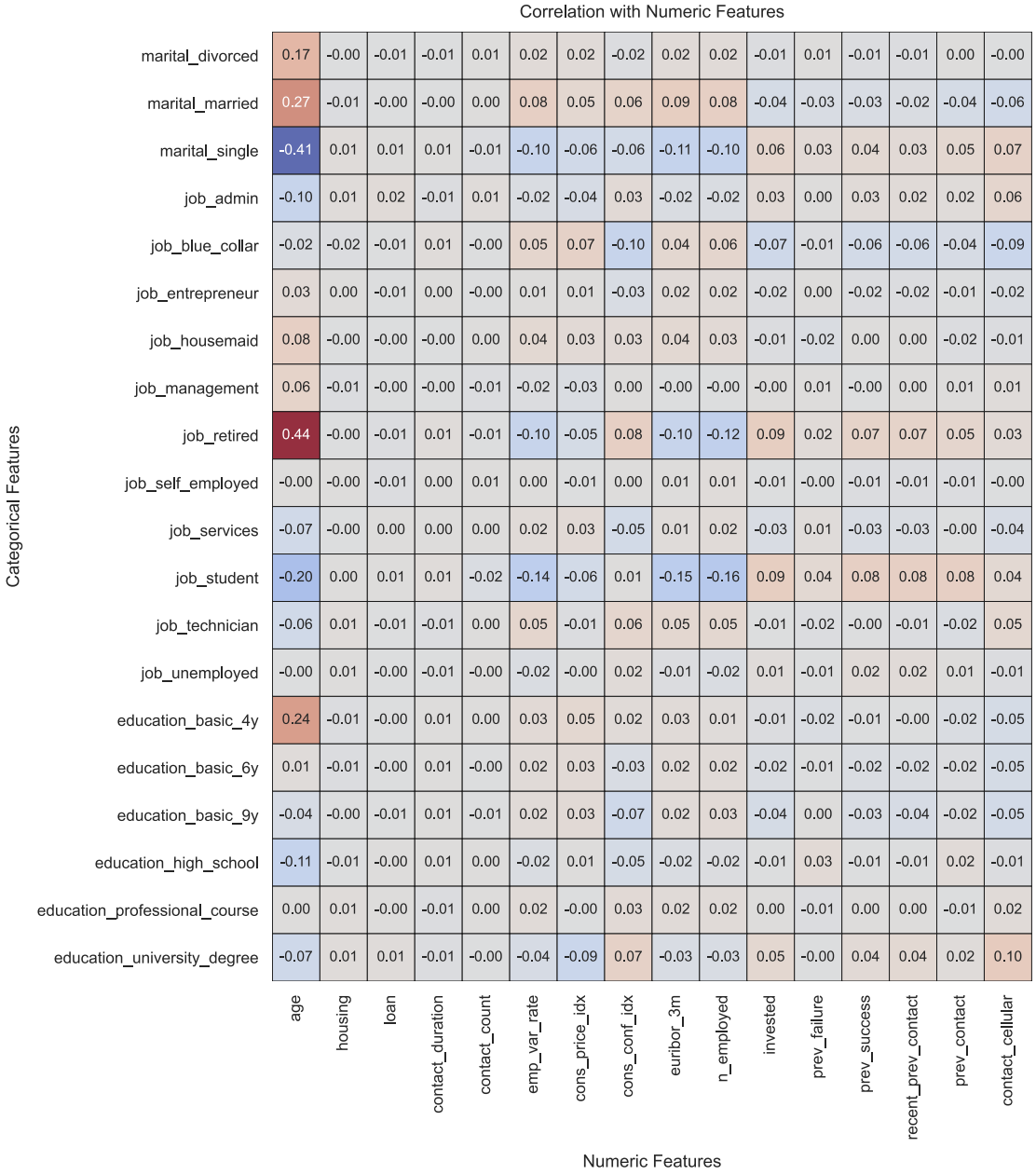



```
<IPython.core.display.Javascript object>
```

The jobs, education levels, and marital statuses seem more interestingly related to 'age' than anything else. There are some weaker relationships here as well.

[29]:
```
plotting.cat_corr_heatmap(data=df,
                          categorical=["marital", "job", "education"],
                          scale=.6,
                          fmt=".2f")
```

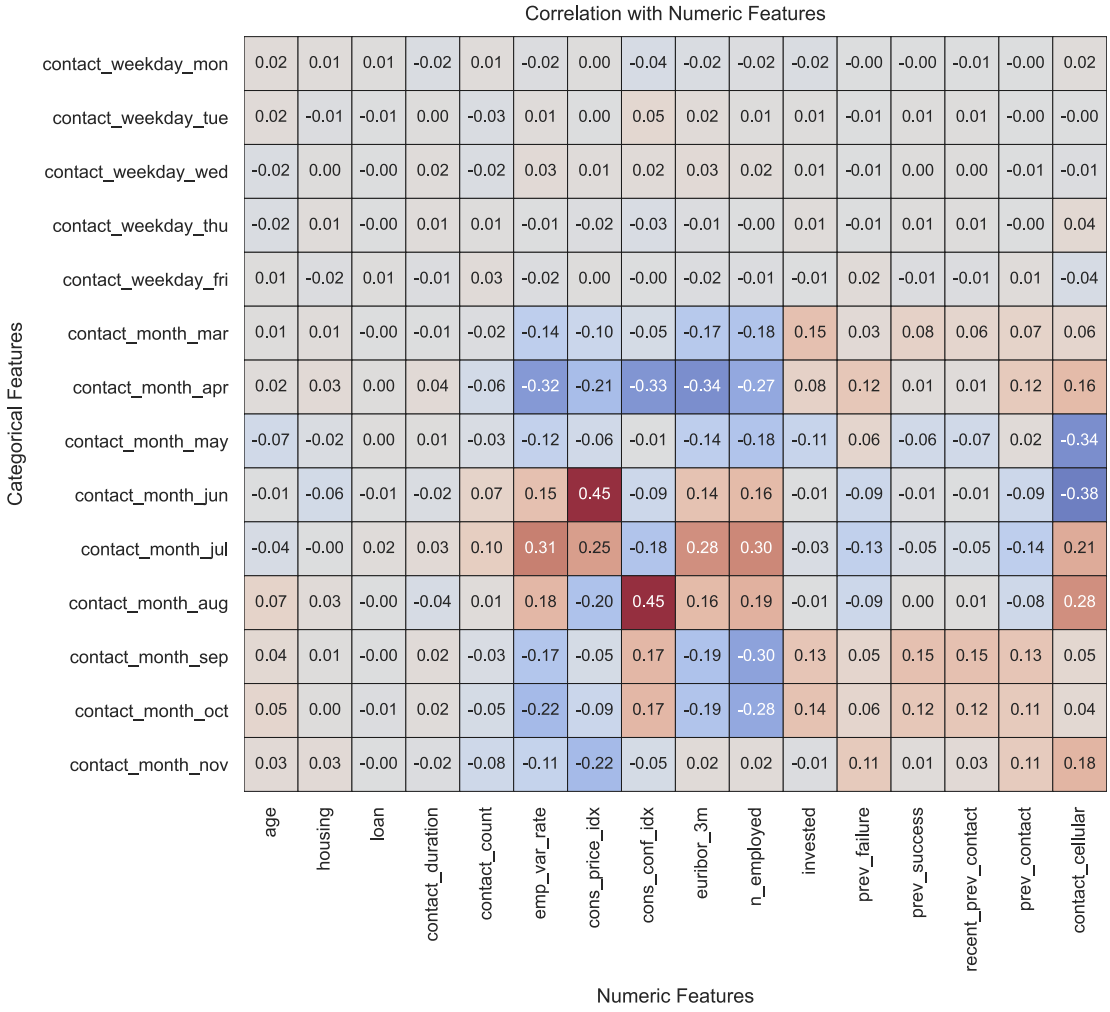
[29]:
```
<AxesSubplot:title={'center':'Correlation with Numeric Features'},
xlabel='Numeric Features', ylabel='Categorical Features'>
```




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

Unsurprisingly there are relatively high correlations between months and economic context variables. After all, the context variables are monthly or quarterly indicators.

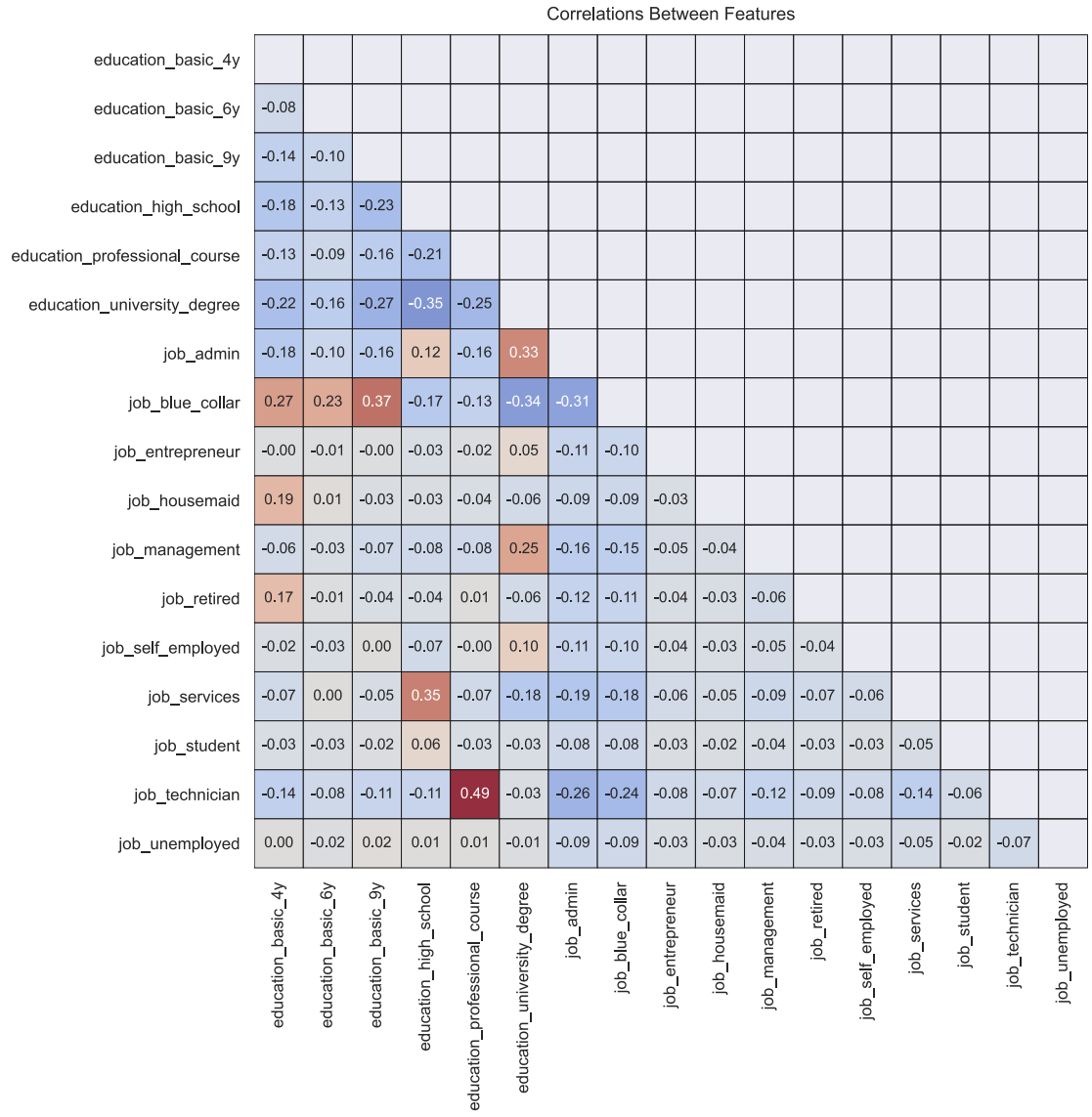
[30]:
```
temporal = ["contact_weekday", "contact_month"]
plotting.cat_corr_heatmap(data=df, categorical=temporal, scale=.6, fmt=".2f")
del temporal
```




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

It's not surprising to find correlations between education types and jobs. The hottest correlation is between 'job_technician' and 'education_professional_course'. Similarly, 'job_services' is highly correlated with 'education_high_school'.

[31]:
```
dummies = pd.get_dummies(df[["education", "job"]])
plotting.pair_corr_heatmap(data=dummies, scale=.6, fmt=".2f")
del dummies
```



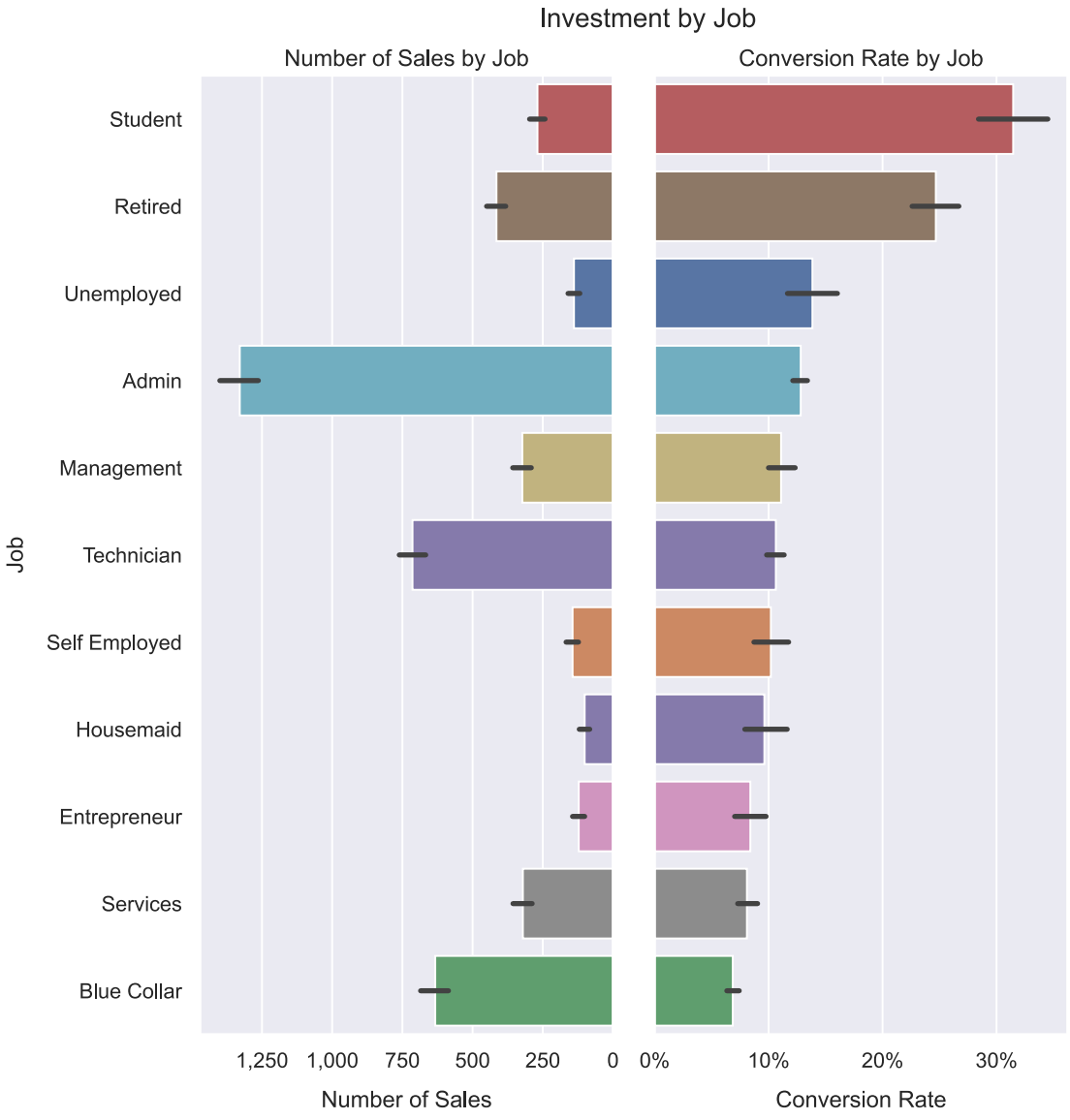

```
<IPython.core.display.Javascript object>
```

[32]:
```
def conversion_bars(*, data, y,  x="invested", title=None, title_pos=(0.6, 1.
↪02), **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}")

    fig.tight_layout()

    if title is not None:
        fig.suptitle(title, x=title_pos[0], y=title_pos[1], fontsize=14)

    return fig
```

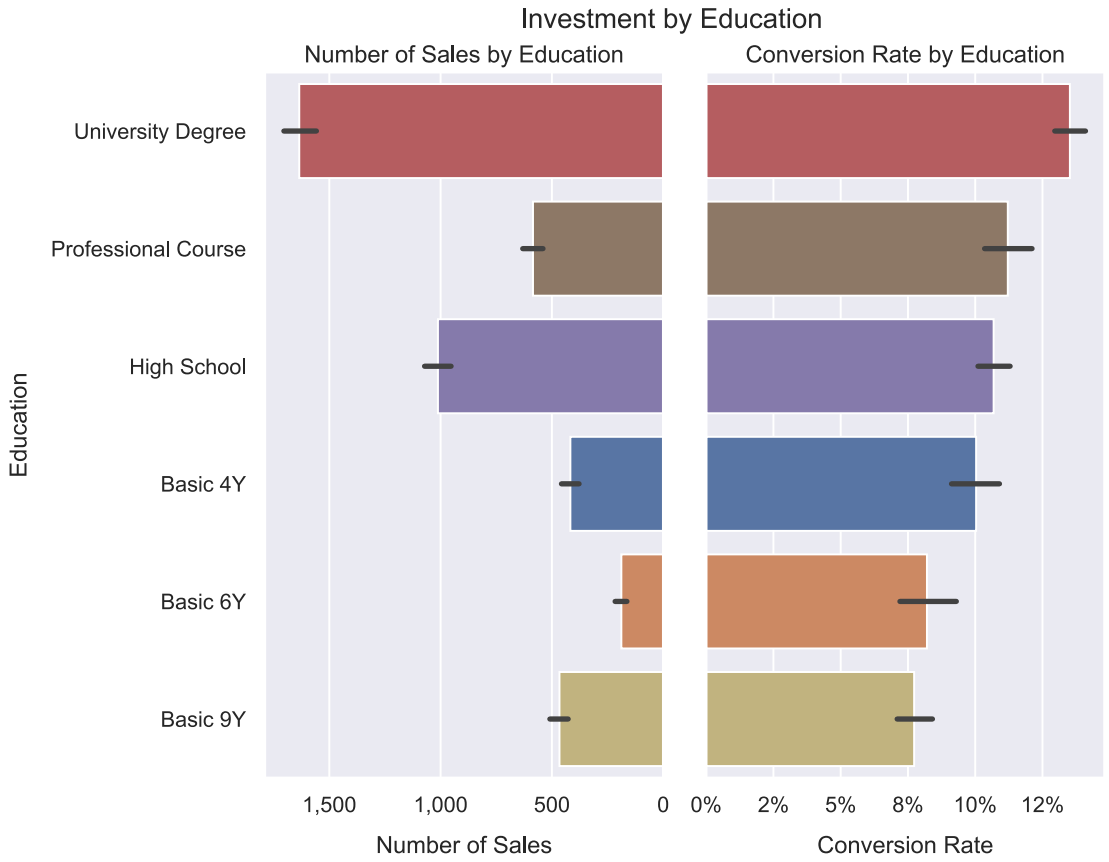
```
<IPython.core.display.Javascript object>
```

[136]:
```
conversion_bars(data=df, y="job", size=(4,8), title="Investment by Job");
```



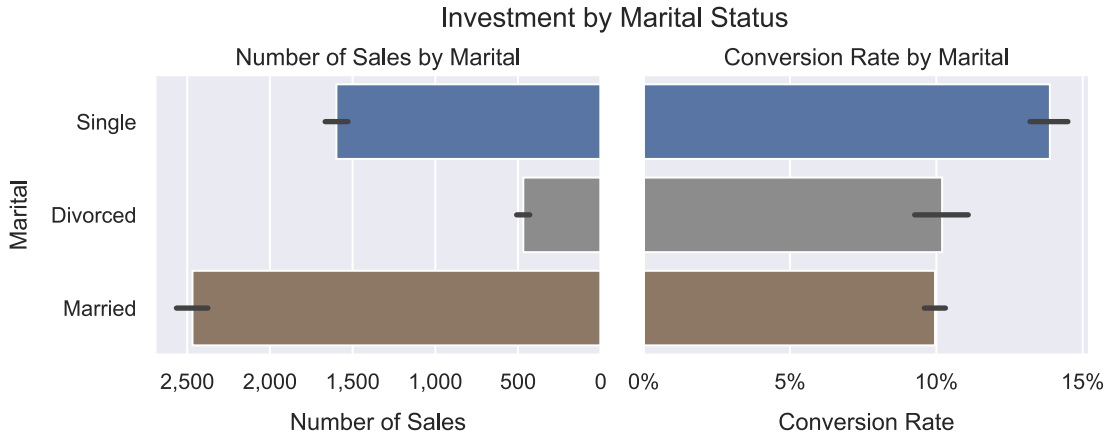

```
<IPython.core.display.Javascript object>
```

[137]:
```
conversion_bars(data=df, x="invested", y="education", title="Investment by␣
↪Education", size=(4, 6));
```




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

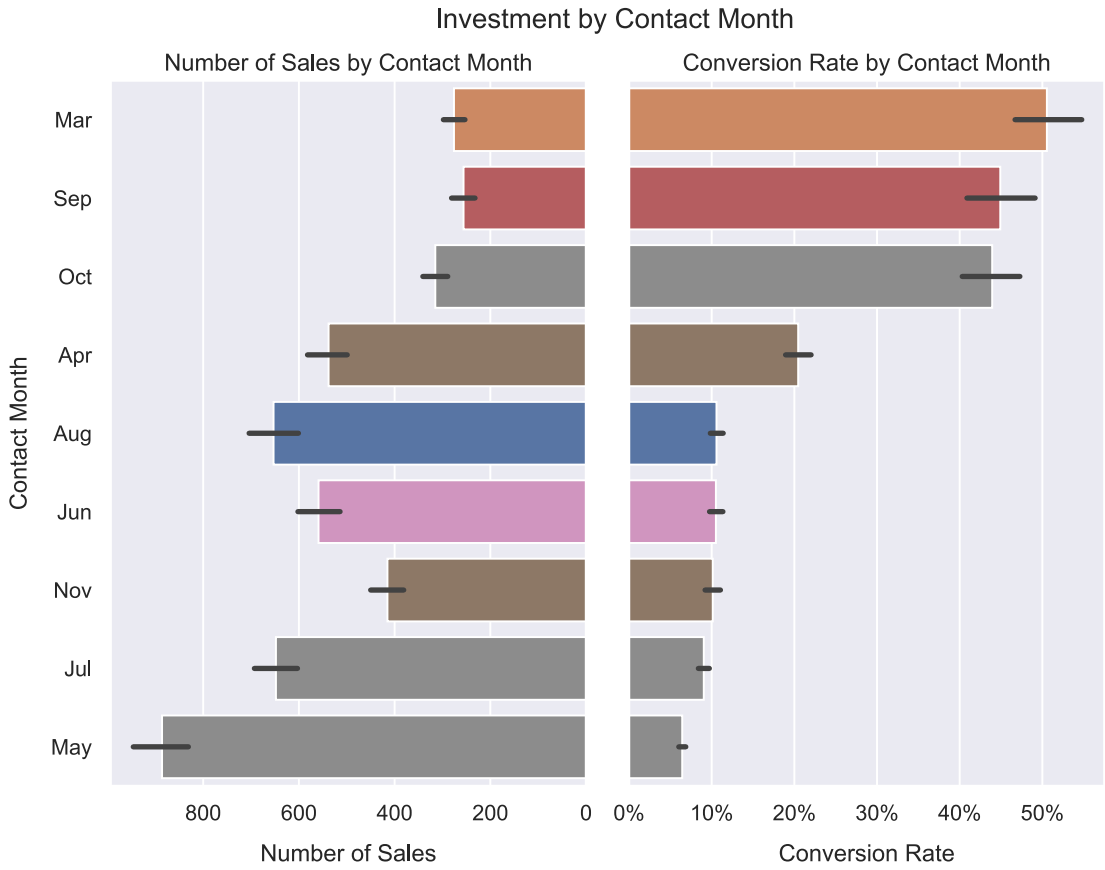
[142]:
```
conversion_bars(data=df, y="marital", title="Investment by Marital Status",␣
↪title_pos=(0.55, 1.05), size=(4, 3));
```



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

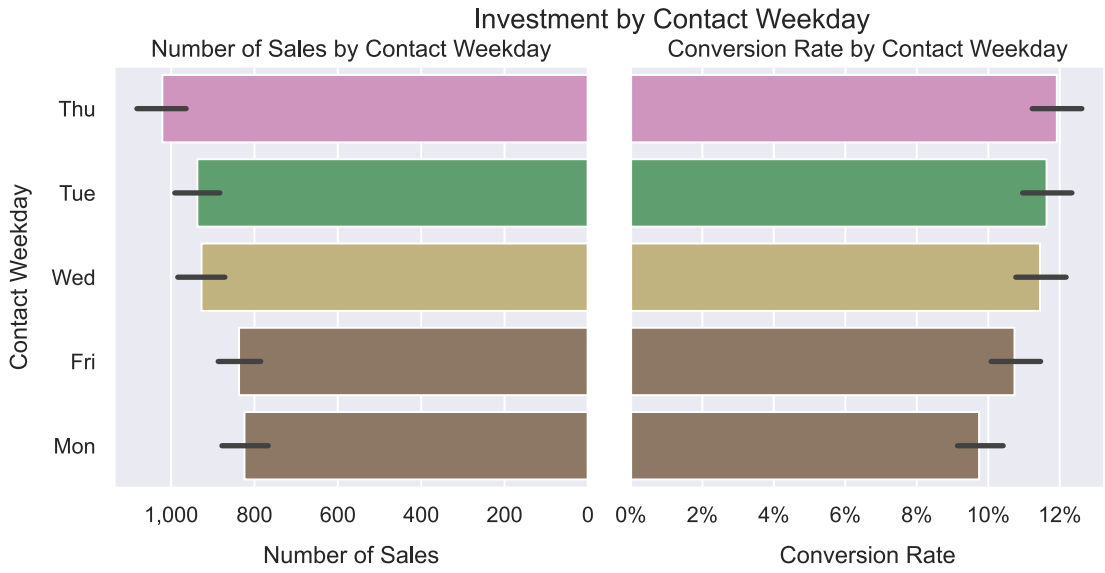
Interestingly, the conversion rate peaks in March, October, and September only to drop low for the remaining months. The number of sales seems almost inversely proportional to the conversion rate. This leaves me unsure as to the ultimate upshot of this graph for Banco de Portugal.

[193]:
```
conversion_bars(data=df, y="contact_month", title="Investment by Contact␣
↪Month", title_pos=(0.55, 1.03), size=(4, 6));
```



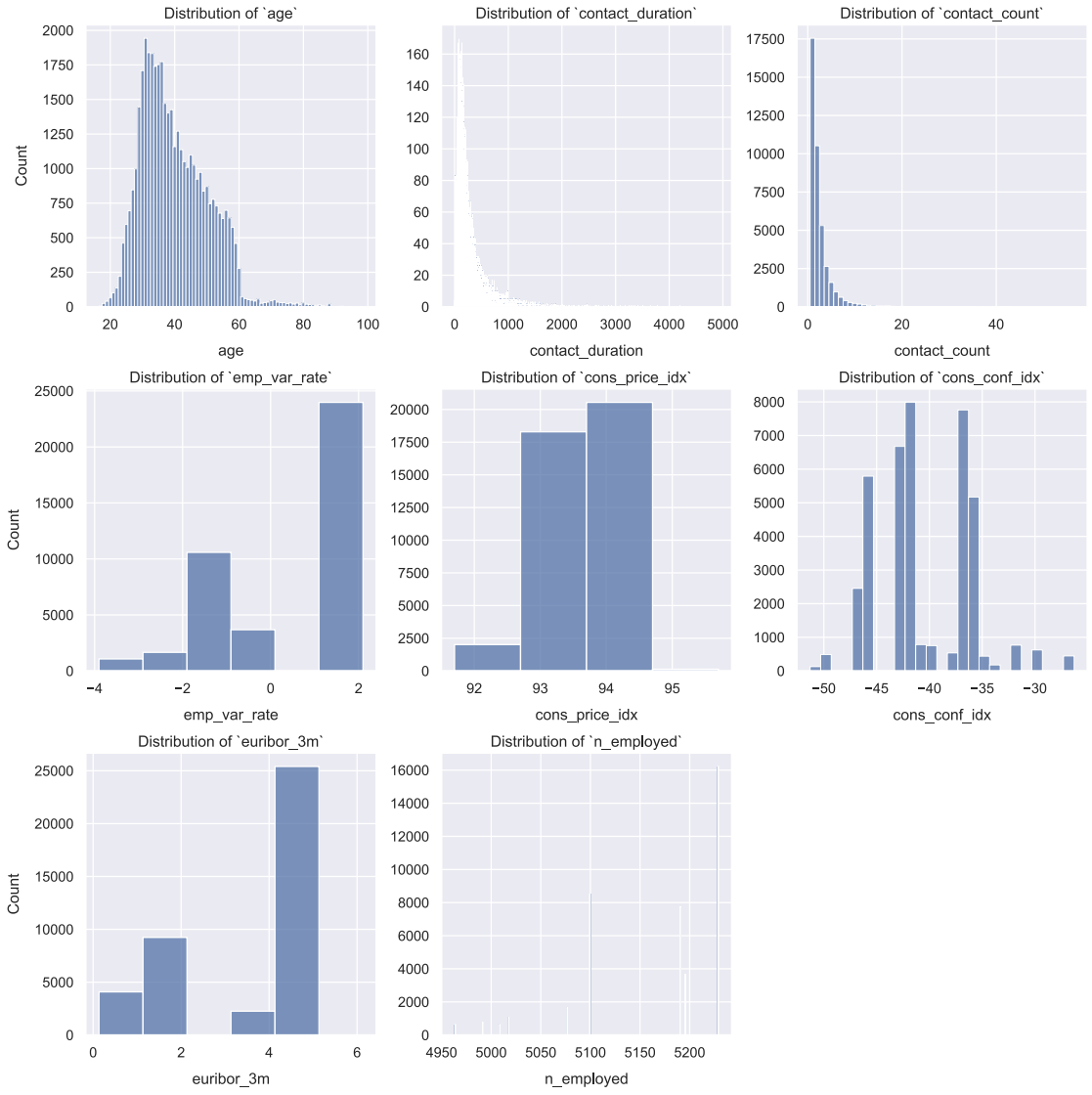

```
<IPython.core.display.Javascript object>
```

[194]:
```
conversion_bars(data=df, y="contact_weekday", title="Investment by Contact␣
↪Weekday", size=(4, 4));
```




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

[149]:
```
plotting.multi_dist(data=df[utils.true_numeric_cols(df)],
                    discrete=True,
                    height=4);
```



<IPython.core.display.Javascript object>

6 Modeling

6.0.1 Modeling Imports

[195]:
```
import joblib
from feature_engine.outliers import Winsorizer
from feature_engine.imputation import ArbitraryNumberImputer
from feature_engine.wrappers import SklearnTransformerWrapper
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, FunctionTransformer
```

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

My classes/modules:

[151]:
```
from tools.modeling.preprocessing import DummyEncoder, FloatArrayForcer
from tools.modeling.selection import SmartCorrelatedSelection
from tools.modeling import selection
```

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

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.

[152]:
```
# 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
```

[152]:
```
((30732, 19), (10244, 19), (30732,), (10244,))
```

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

6.1 First Model

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

6.1.1 Baseline Preprocessors

All of the preprocessors except `FloatArrayForcer` take and return DataFrames rather than NDarrays, which are standard for Scikit-Learn. The advantage of using DataFrames during preprocessing is that the features are labeled and easily accessible. `FloatArrayForcer` returns a float array, and is naturally the last preprocessing step before the final estimator.

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-Learn `OneHotEncoder`.

SmartCorrelatedSelection I use a feature selection tool from the Feature Engine package which detects highly correlated sets of features and keeps only one, which it chooses intelligently. The purpose is to avoid multicollinearity, which is a potential problem in logistic regression. If features are highly 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. >Note: I created a wrapper for this which displays the results every time `fit()` is called.

ArbitraryNumberImputer 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 (i.e. False) during one-hot encoding. There is not really a more sophisticated way to handle those, except perhaps filling with mode. I use `ArbitraryNumberImputer` from Feature Engine to fill missing values with 0 in the remaining two binary categoricals, 'housing' and 'loan'. This is equivalent to filling with mode, because the mode is 0 for both of those. Filling missing values with 0 is fast, and there aren't enough missing values to justify using a more sophisticated method.

FloatArrayForcer As the last preprocessing step before the final estimator, I coerce the data into a float array using my wrapper for `sklearn.utils.as_float_array`. Since `DummyEncoder` and the transformers from Feature Engine work with DataFrames that may contain different dtypes, it's handy explicitly force the data into a float array. This is probably unnecessary, since Scikit-Learn estimators usually coerce the data into a float array anyway. Nevertheless, it's nice to store all the DataFrame feature names in the last transformer for easy access.

As a reminder, the pipeline must handle the following missing values:

[153]:
```
cleaning.info(df).head(6).style.bar(subset=["nan"])
```

[153]:
```
<pandas.io.formats.style.Styler at 0x1f4717eedc0>

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

Baseline Classifier Pipeline

[154]:
```
corr_trimmer = SmartCorrelatedSelection(
    selection_method="variance", verbose=True)
corr_trimmer
```

[154]:
```
SmartCorrelatedSelection(selection_method='variance', verbose=True)

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

[155]:
```
classifier_pipe = Pipeline([
    ("cat_encoder", DummyEncoder(drop_first=True)),
    ("corr_trimmer", corr_trimmer),
    ("imputer", ArbitraryNumberImputer(0)),
```

```
    ("float_forcer", FloatArrayForcer()),
    ("classifier", DummyClassifier())
])

classifier_pipe
```

[155]:
```
Pipeline(steps=[('cat_encoder', DummyEncoder(drop_first=True)),
                ('corr_trimmer',
                 SmartCorrelatedSelection(selection_method='variance',
                                          verbose=True)),
                ('imputer', ArbitraryNumberImputer(arbitrary_number=0)),
                ('float_forcer', FloatArrayForcer()),
                ('classifier', DummyClassifier())])
```

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

Here is the output of the first 3 preprocessing steps, just before the data is coerced into a float array. Notice the new columns added by `DummyEncoder`.

Notice also the features that were dropped because of multicollinearity concerns. My `corr_trimmer` kept 'prev_contact', 'prev_success', 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.

[156]:
```
df_out = classifier_pipe[:3].fit_transform(X_train, y_train)
df_out.head()
```

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

[156]:

	age	housing	loan	contact_count	cons_price_idx	cons_conf_idx	\
27532	32	1.0	1.0	3	93.200	-42.0	
18595	39	0.0	0.0	9	93.918	-42.7	
27281	50	1.0	0.0	2	93.200	-42.0	
8870	32	1.0	0.0	1	94.465	-41.8	
38500	30	1.0	0.0	2	92.431	-26.9	

	n_employed	prev_success	prev_contact	contact_cellular	…	\
27532	5195.8	0.0	0.0	1.0	…	
18595	5228.1	0.0	0.0	1.0	…	
27281	5195.8	0.0	0.0	0.0	…	
8870	5228.1	0.0	0.0	0.0	…	
38500	5017.5	0.0	0.0	0.0	…	

	contact_month_jun	contact_month_jul	contact_month_aug	\
27532	0.0	0.0	0.0	
18595	0.0	1.0	0.0	
27281	0.0	0.0	0.0	
8870	1.0	0.0	0.0	
38500	0.0	0.0	0.0	

```
       contact_month_sep  contact_month_oct  contact_month_nov  \
27532                0.0                0.0                1.0
18595                0.0                0.0                0.0
27281                0.0                0.0                1.0
8870                 0.0                0.0                0.0
38500                0.0                1.0                0.0

       contact_weekday_tue  contact_weekday_wed  contact_weekday_thu  \
27532                  0.0                  0.0                  0.0
18595                  0.0                  0.0                  1.0
27281                  0.0                  0.0                  0.0
8870                   0.0                  0.0                  1.0
38500                  1.0                  0.0                  0.0

       contact_weekday_fri
27532                  1.0
18595                  0.0
27281                  1.0
8870                   0.0
38500                  0.0

[5 rows x 39 columns]
```

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

Here is the float array spit out by the first 4 steps. You can recognize the 'age' column on the left.

[157]:
```
np_out = classifier_pipe[:4].fit_transform(X_train, y_train)
np_out
```

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

[157]:
```
array([[32.,  1.,  1., …,  0.,  0.,  1.],
       [39.,  0.,  0., …,  0.,  1.,  0.],
       [50.,  1.,  0., …,  0.,  0.,  1.],
       …,
       [27.,  0.,  0., …,  0.,  0.,  0.],
       [40.,  1.,  0., …,  1.,  0.,  0.],
       [35.,  0.,  0., …,  0.,  0.,  0.]])
```

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

And here is the confirmation that the DataFrame output is equivalent to the NDarray output. That's enough preprocessing for now—time to move on to the dummy model.

[158]:
```
# Show that DataFrame equals float array
display(np.array_equal(df_out, np_out))

# Mop up variables
del df_out, np_out
```

```
True
```

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

6.1.2 Dummy Model

I create a dummy model which makes random predictions weighted by the class distribution ratio. 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.

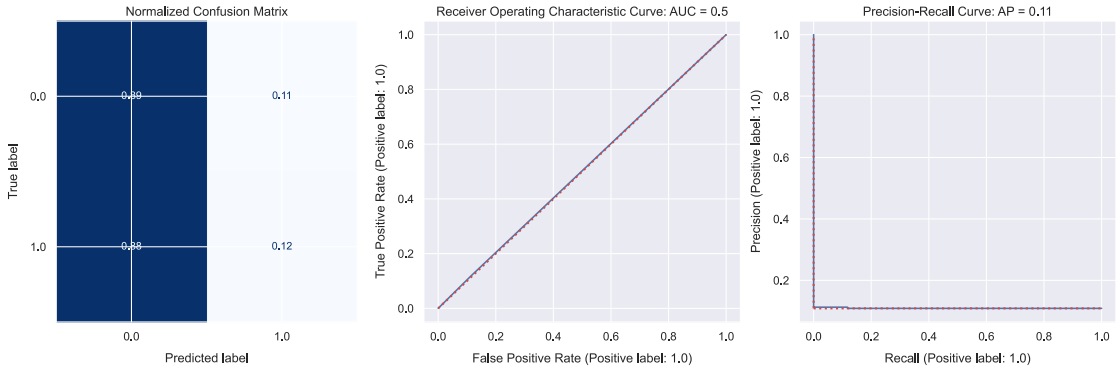
The confusion matrix indicates that the dummy gets 90% of the true negatives and 12% of the true positives, which is pretty bad.

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 ~0.5.

[159]:
```
classifier_pipe["classifier"].set_params(strategy="stratified", random_state=60)
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test, zero_division=0)
```

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

```
<pandas.io.formats.style.Styler at 0x1f4760302e0>
```




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

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.

[160]:
```
logit = LogisticRegression(fit_intercept=False,
                           penalty="none",
                           multi_class="ovr",
                           max_iter=1e4,
                           warm_start=False,
```

```
                              solver="lbfgs")
logit
```

[160]:
```
LogisticRegression(fit_intercept=False, max_iter=10000.0, multi_class='ovr',
                   penalty='none')
```

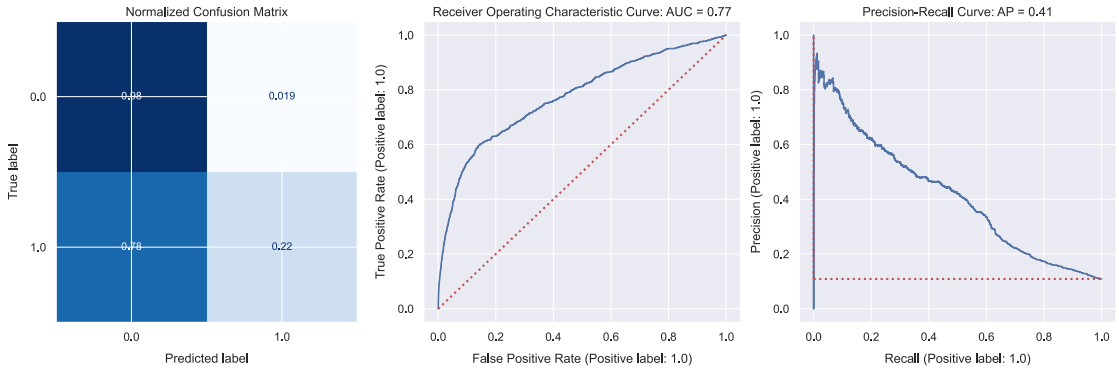
```
<IPython.core.display.Javascript object>
```

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.

[161]:
```
# Set classifier (final step) to `logit`
classifier_pipe.set_params(classifier=logit)

# Train and test
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

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

```
<pandas.io.formats.style.Styler at 0x1f4774b5640>
```



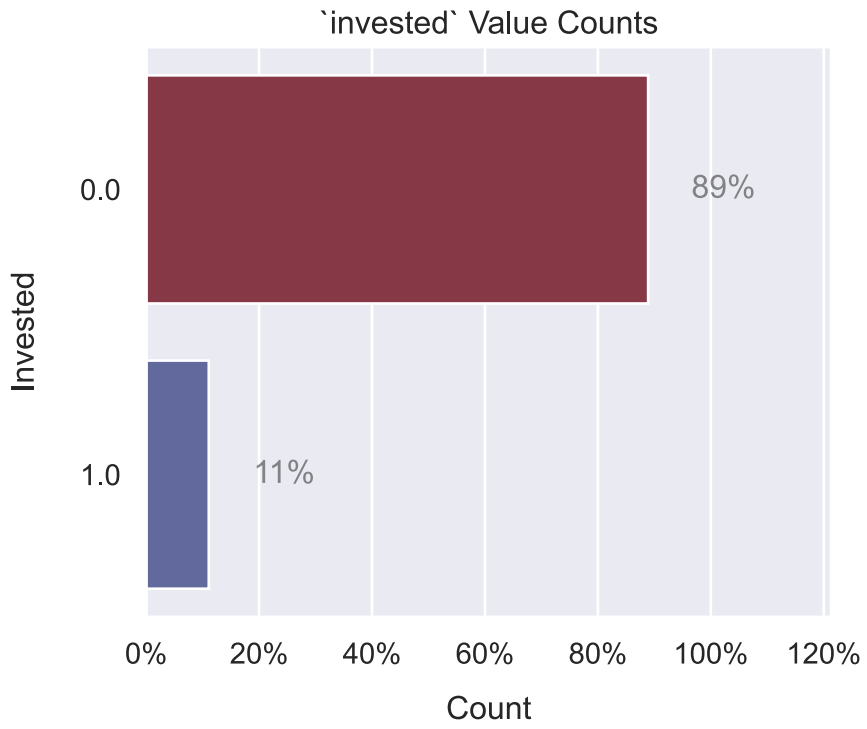

```
<IPython.core.display.Javascript object>
```

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.

[162]:
```
fig = plotting.multi_countplot(data=y.to_frame(),
                               normalize=True,
                               orient="h",
                               height=4)
```




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

[163]:
```
# Set classifier to use balanced class weights
logit.set_params(class_weight="balanced")

# Display pipeline
classifier_pipe
```

[163]:
```
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…
                                                'education_basic_6y',
                                                'education_basic_9y',
                                                'education_high_school',
 'education_professional_course',
 'education_university_degree',
                                                'contact_month_apr',
                                                'contact_month_may',
                                                'contact_month_jun', …])),
                 ('float_forcer', FloatArrayForcer()),
                 ('classifier',
                  LogisticRegression(class_weight='balanced',
                                     fit_intercept=False, max_iter=10000.0,
                                     multi_class='ovr', penalty='none'))])
```

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

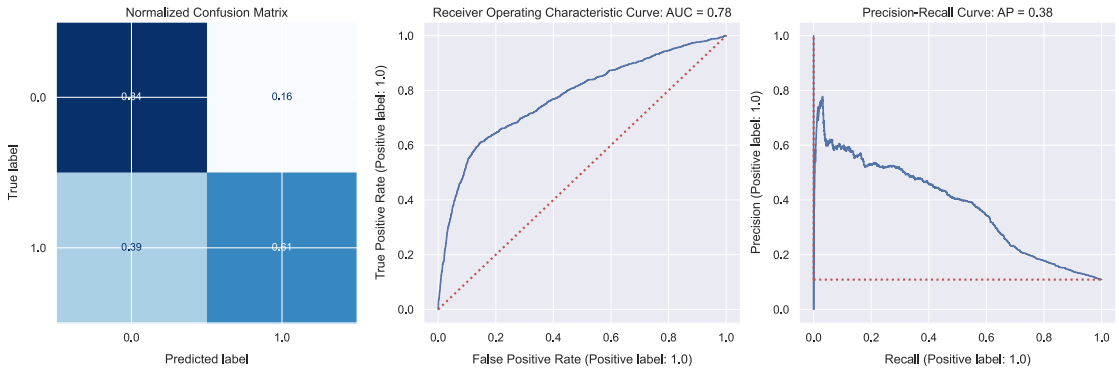
6.2.2 Train and Test

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.

[164]:
```
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

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

```
<pandas.io.formats.style.Styler at 0x1f4761786a0>
```




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

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.

- I scale after one-hot encoding because from the algorithm's perspective, one-hot encoded categoricals are no different than any other numeric variable, and I want everything to be on the same scale.
- I scale before imputation so that the imputed values don't distort the mean.

[165]:
```
classifier_pipe = Pipeline([
    ("cat_encoder", DummyEncoder(drop_first=True)),
    ("corr_trimmer", corr_trimmer),

    # Wrap StandardScaler() so it returns DataFrame
    ("scaler", SklearnTransformerWrapper(transformer=StandardScaler())),

    ("imputer", ArbitraryNumberImputer(0)),
    ("float_forcer", FloatArrayForcer()),
    ("classifier", logit)
])

classifier_pipe
```

[165]:
```
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…
                                                     'education_high_school',
 'education_professional_course', …],
                                          verbose=True)),
                ('scaler',
                 SklearnTransformerWrapper(transformer=StandardScaler())),
                ('imputer', ArbitraryNumberImputer(arbitrary_number=0)),
```

```
                ('float_forcer', FloatArrayForcer()),
                ('classifier',
                 LogisticRegression(class_weight='balanced',
                                    fit_intercept=False, max_iter=10000.0,
                                    multi_class='ovr', penalty='none'))])
```

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

The preprocessing output looks appropriately scaled.

[166]:
```
classifier_pipe[:4].fit_transform(X_train, y_train).head()
```

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

[166]:

	age	housing	loan	contact_count	cons_price_idx	\
27532	-0.771218	0.929451	2.334618	0.149630	-0.659931	
18595	-0.095600	-1.075904	-0.428336	2.311735	0.582839	
27281	0.966086	0.929451	-0.428336	-0.210721	-0.659931	
8870	-0.771218	0.929451	-0.428336	-0.571071	1.529628	
38500	-0.964252	0.929451	-0.428336	-0.210721	-1.990975	

	cons_conf_idx	n_employed	prev_success	prev_contact	\
27532	-0.315351	0.389430	-0.184152	-0.395341	
18595	-0.467204	0.839096	-0.184152	-0.395341	
27281	-0.315351	0.389430	-0.184152	-0.395341	
8870	-0.271964	0.839096	-0.184152	-0.395341	
38500	2.960333	-2.092785	-0.184152	-0.395341	

	contact_cellular	…	contact_month_jun	contact_month_jul	\
27532	0.760811	…	-0.386157	-0.462221	
18595	0.760811	…	-0.386157	2.163467	
27281	-1.314386	…	-0.386157	-0.462221	
8870	-1.314386	…	2.589617	-0.462221	
38500	-1.314386	…	-0.386157	-0.462221	

	contact_month_aug	contact_month_sep	contact_month_oct	\
27532	-0.420416	-0.120382	-0.133358	
18595	-0.420416	-0.120382	-0.133358	
27281	-0.420416	-0.120382	-0.133358	
8870	-0.420416	-0.120382	-0.133358	
38500	-0.420416	-0.120382	7.498603	

	contact_month_nov	contact_weekday_tue	contact_weekday_wed	\
27532	2.996318	-0.493616	-0.497284	
18595	-0.333743	-0.493616	-0.497284	
27281	2.996318	-0.493616	-0.497284	
8870	-0.333743	-0.493616	-0.497284	
38500	-0.333743	2.025867	-0.497284	

```
       contact_weekday_thu  contact_weekday_fri
27532            -0.513672             2.060187
18595             1.946767            -0.485393
27281            -0.513672             2.060187
8870              1.946767            -0.485393
38500            -0.513672            -0.485393

[5 rows x 39 columns]
```

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

I set the solver to 'saga' now that I'm using scaled data. I also reduce the max iterations to 1,000 because the algorithm converges faster when using scaled data. I ensure that regularization ('penalty') is turned off.

[167]:
```
logit.set_params(solver="saga", penalty="none", C=1, max_iter=1e3)
```

[167]:
```
LogisticRegression(C=1, class_weight='balanced', fit_intercept=False,
                   max_iter=1000.0, multi_class='ovr', penalty='none',
                   solver='saga')
```

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

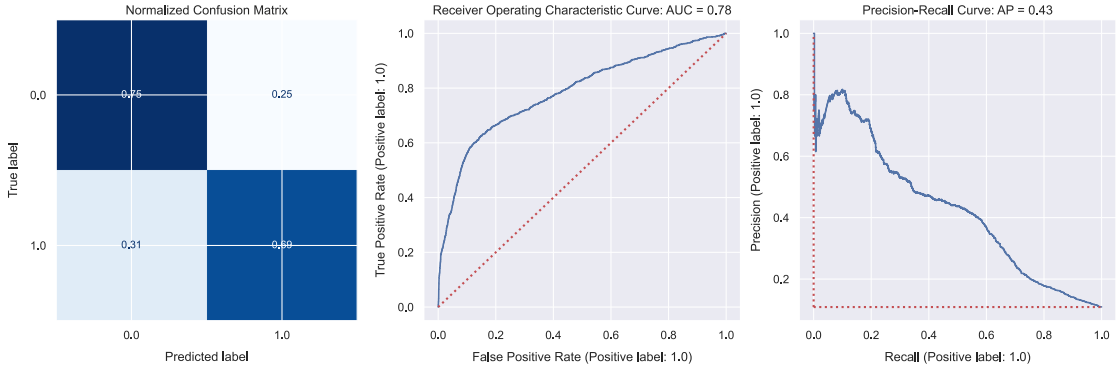
6.3.2 Train and Test

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

[168]:
```
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

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

```
<pandas.io.formats.style.Styler at 0x1f4721da400>
```




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

6.4 Fourth Model

6.4.1 Winsorize before Scaling

Since my distributions are variously skewed and non-normal, I Winsorize them at 90% in order to reduce the influence of outliers on each distribution's mean and make them more suitable for `StandardScaler`. I only Winsorize the genuine numeric features, i.e. those which have at least 3 unique values.

[169]:
```
winsorizer = Winsorizer(variables=utils.true_numeric_cols(X),
                        capping_method="quantiles",
                        tail="both",
                        fold=0.05)
```

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

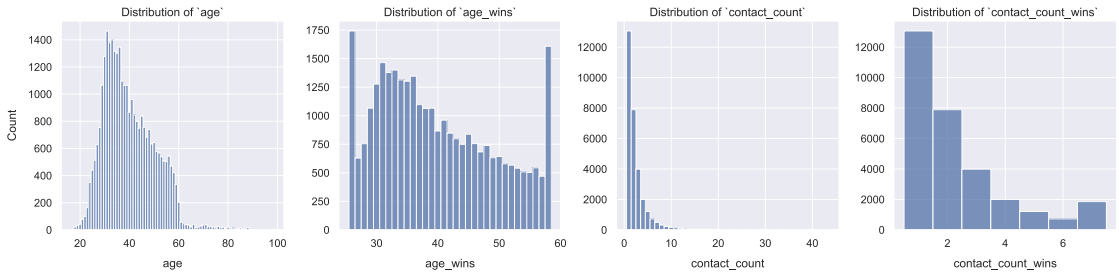
[170]:
```
classifier_pipe = Pipeline([
    ("cat_encoder", DummyEncoder(drop_first=True)),
    ("winsorizer", winsorizer),
    ("corr_trimmer", corr_trimmer),
    ("scaler", SklearnTransformerWrapper(transformer=StandardScaler())),
    ("imputer", ArbitraryNumberImputer(0)),
    ("float_forcer", FloatArrayForcer()),
    ("classifier", logit)
])

classifier_pipe
```

[170]:
```
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',
                                          variables=['age', 'housing', 'loan',
                                                     '…
 'education_professional_course', …],
                                          verbose=True)),
                ('scaler',
                 SklearnTransformerWrapper(transformer=StandardScaler())),
                ('imputer', ArbitraryNumberImputer(arbitrary_number=0)),
                ('float_forcer', FloatArrayForcer()),
                ('classifier',
                 LogisticRegression(C=1, class_weight='balanced',
                                    fit_intercept=False, max_iter=1000.0,
                                    multi_class='ovr', penalty='none',
                                    solver='saga'))])
```

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

To Winsorize a distribution at 90% is to clip it to the outermost values between the 5^{th} and 95^{th} percentiles. As you can see from the examples below, this means moving far-out data points to new locations within the inner 90% range.

[171]:
```
# List features to use for demo
demo_feats = ["age", "contact_count"]

# Get DataFrame of unprocessed demo features
examples = X_train.loc[:, demo_feats].copy()

# Winsorize data and slice out demo features
win_examples = winsorizer.fit_transform(X_train).loc[:, demo_feats]

# Merge unprocessed and Winsorized examples
examples = examples.join(win_examples, rsuffix="_wins").sort_index(axis=1)

# Plot distributions
fig = plotting.multi_dist(data=examples, ncols=4, discrete=True, height=4);
fig.tight_layout()

# Mop up temp variables
del fig, examples, win_examples, demo_feats
```




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

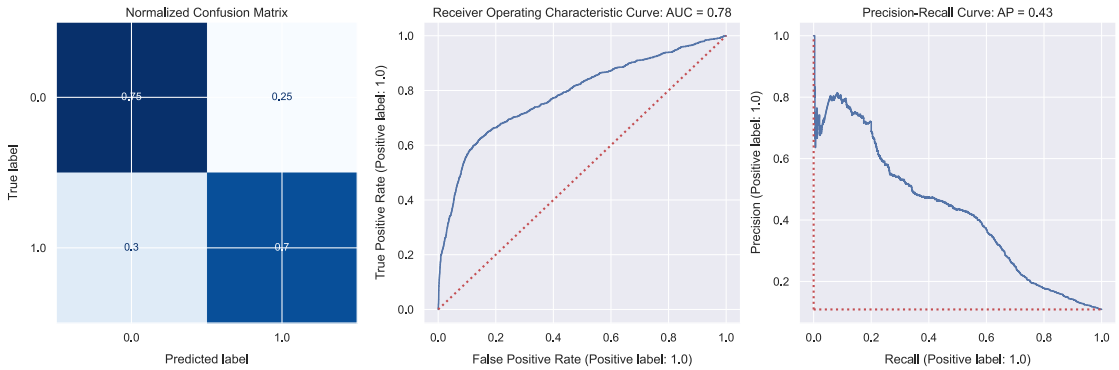
6.4.2 Train and Test

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.

[172]:
```
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

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

```
<pandas.io.formats.style.Styler at 0x1f47787f2b0>
```




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

6.5 Final Model

6.5.1 Hyperparameter Tuning

It would be wise to tune the Winsorization 'fold' parameter alongside the inverse regularization strength C. I set up a parameter grid to run a grid search on `classifier_pipe`. One of my favorite features of Scikit-Learn's `GridSearchCV` is that you can run a search over an entire pipeline.

[173]:
```
grid = dict(classifier__C=np.geomspace(1e-5, 1e5, 11),
            winsorizer__fold=np.linspace(.0, .2, 5))
grid
```

[173]:
```
{'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 ])}
```

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

I set the `logit` regularization to L2 in preparation for the search.

[174]:
```
logit.set_params(penalty="l2")
```

[174]:
```
LogisticRegression(C=1, class_weight='balanced', fit_intercept=False,
                   max_iter=1000.0, multi_class='ovr', solver='saga')
```

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

I create the `GridSearchCV` object using 5-fold cross-validation repeated 10× to reduce the effect of random chance on the results. I choose *average precision*, *ROC AUC (weighted)*, and *balanced accuracy* as metrics. These three metrics summarize the *precision-recall curve*, the *receiver operating characteristic curve*, and the overall accuracy.

[175]:
```
# 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"]

# Search object -- no need to refit with best params
search = GridSearchCV(classifier_pipe,
                      grid,
                      n_jobs=-1,
                      cv=validator,
                      scoring=scoring,
                      refit=False)


search
```

[175]:
```
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'])

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

Here I run the actual search. The following code block contains the code for running the search. It takes ~8 minutes to run if turned back into a code cell.

```
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()
```

[176]:
```
# 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)
```

[176]:

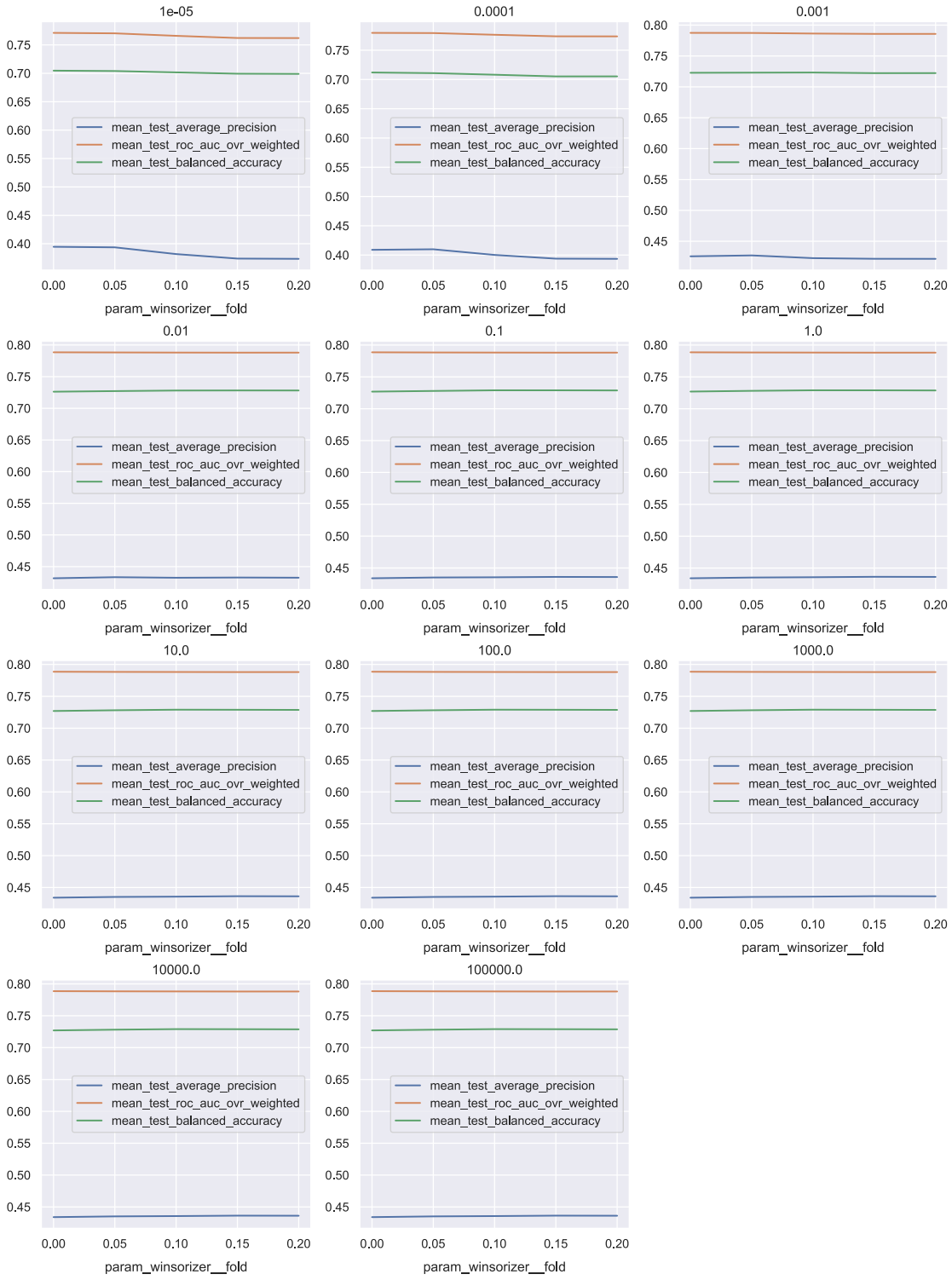
	param_classifier__C	param_winsorizer__fold	mean_test_average_precision	\
0	0.00001	0.0	0.394610	
30	10.00000	0.0	0.434002	
50	100000.00000	0.0	0.434003	
5	0.00010	0.0	0.408939	
20	0.10000	0.0	0.433878	

	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

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

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

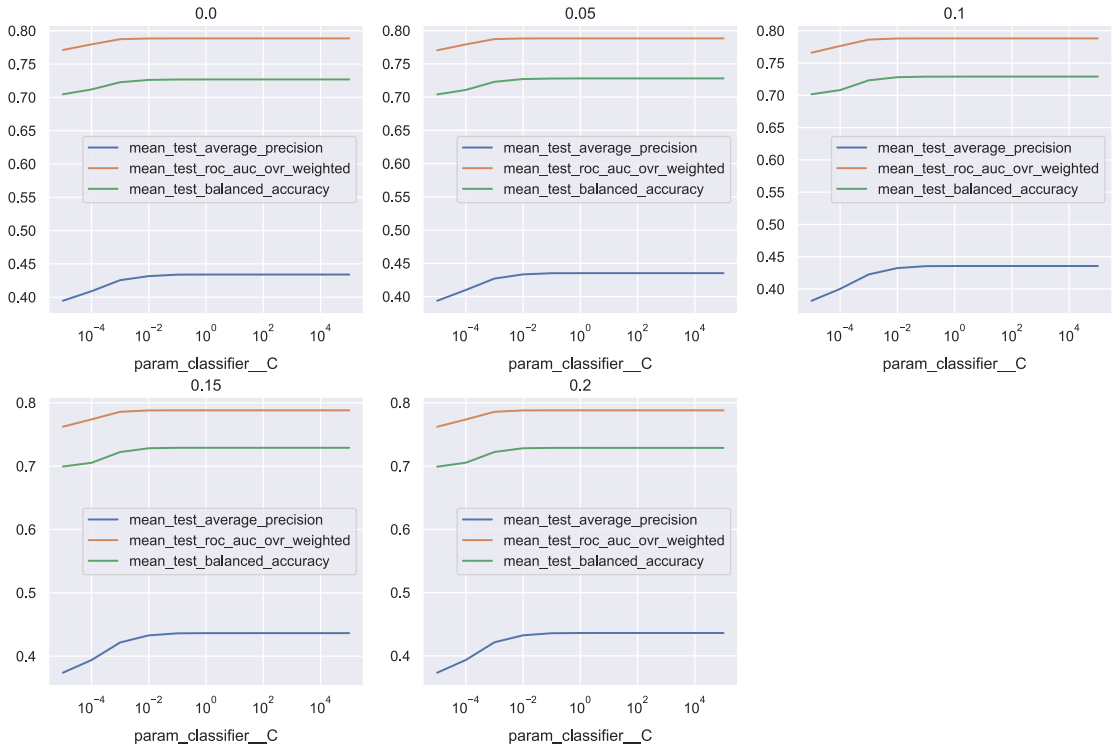
[177]:
```
plotting.grouper_plot(data=results,
                      grouper="param_classifier__C",
                      x="param_winsorizer__fold",
                      y=scores);
```



<IPython.core.display.Javascript object>

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

[178]:
```
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")
```



<IPython.core.display.Javascript object>

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 'fold' 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.

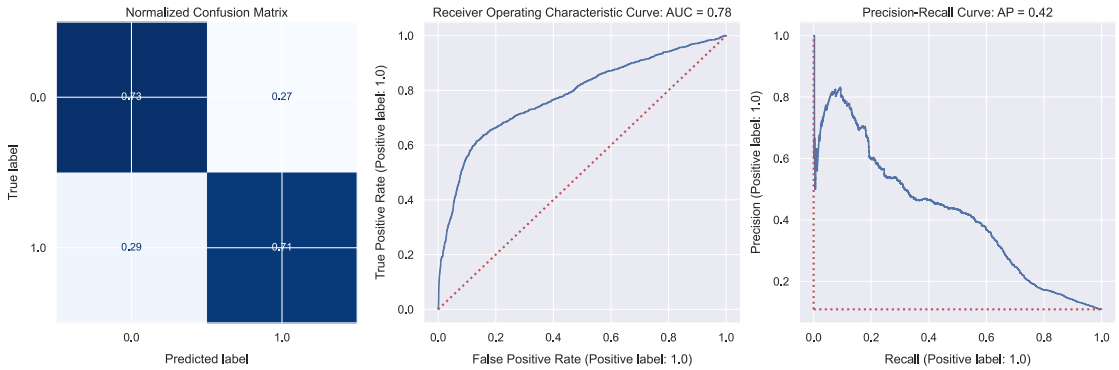
[179]:
```
# Turn off warm start, set regularization to L2 at C=0.001
logit.set_params(warm_start=False, penalty="l2", C=1e-3)
display(logit)

# Train and test
```

```
classifier_pipe.fit(X_train, y_train)
diagnostics.standard_report(classifier_pipe, X_test, y_test)
```

```
LogisticRegression(C=0.001, class_weight='balanced', fit_intercept=False,
                   max_iter=1000.0, multi_class='ovr', solver='saga')
```

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

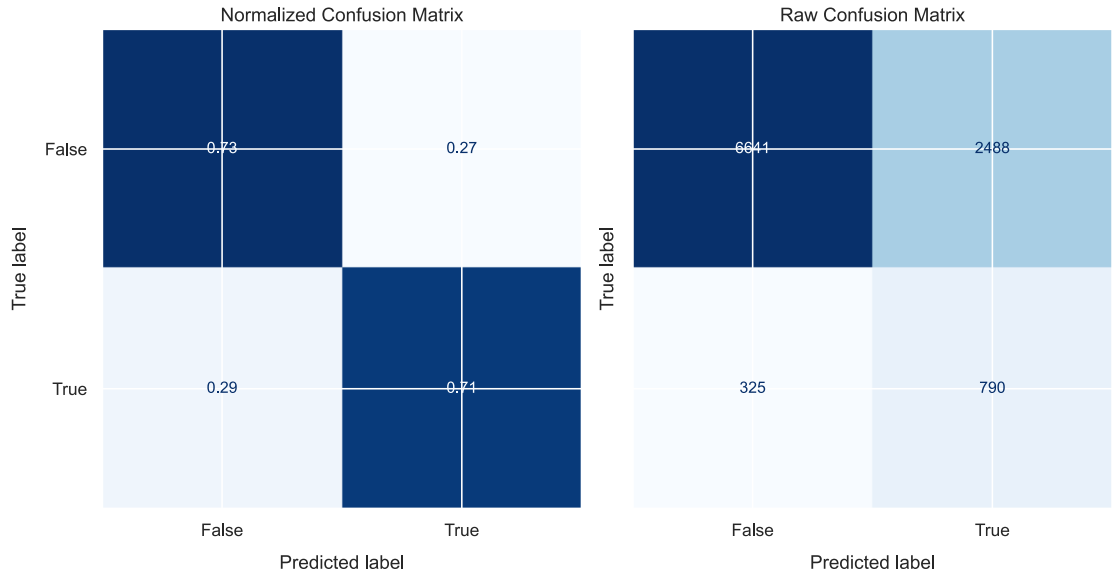
```
<pandas.io.formats.style.Styler at 0x1f46beef100>
```




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

Both normalized and raw confusion matrices for the presentation.

[180]:
```
diagnostics.plot_double_confusion_matrices(
    classifier_pipe, X_test, y_test, display_labels=[False, True]);
```



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

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.

[181]:
```
classifier_pipe.fit(X, y)
joblib.dump(classifier_pipe, normpath("models/final_model.joblib"))
```

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

[181]:
```
['models\\final_model.joblib']
```

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

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 `FloatArrayForcer`.

[182]:
```
feat_names = classifier_pipe["float_forcer"].feature_names_
feat_names
```

[182]:
```
array(['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'], dtype=object)
```

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

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

[183]:
```
coef = pd.DataFrame(logit.coef_.T,
                    columns=["coef"],
                    index=feat_names).squeeze().copy()
print(f"size: {coef.size}")
coef.sort_values().head()
```

```
size: 39
```

[183]:
```
n_employed          -0.624925
contact_month_may   -0.288578
contact_month_nov   -0.090779
prev_contact        -0.081289
contact_month_aug   -0.064638
Name: coef, dtype: float64
```

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

All Coefficients

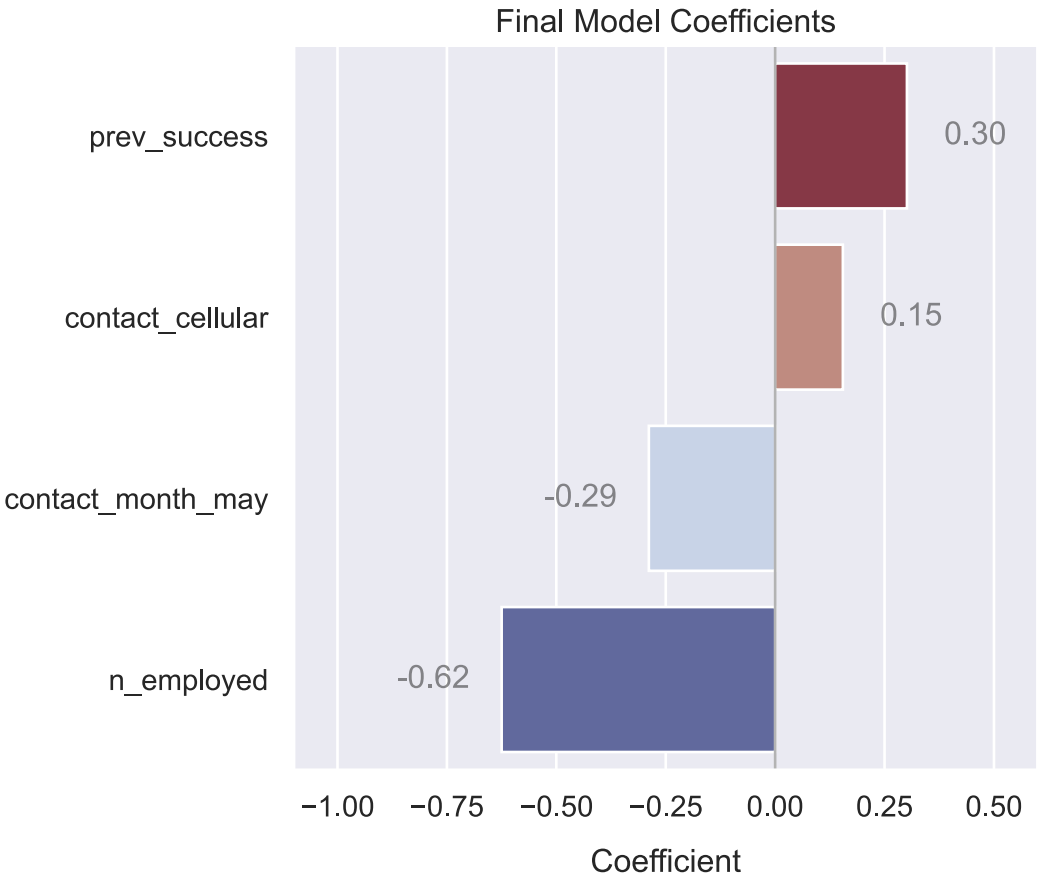
[184]:
```
ax = plotting.heated_barplot(data=coef,
                             figsize=(8, 15))
plotting.annot_bars(ax, dist=.15)
ax.set(xlabel="Coefficient", title="Final Model: All Coefficients");
```


Final Model: All Coefficients
prev_success
contact_cellular
cons_conf_idx
job_retired
contact_month_oct
contact_weekday_wed
job_student
contact_month_jun
education_university_degree
contact_month_jul
contact_weekday_thu
contact_weekday_fri
contact_weekday_tue
marital_single
education_basic_6y
education_professional_course
education_high_school
contact_month_apr
job_entrepreneur
job_management
job_unemployed
marital_married
education_basic_9y
contact_month_sep
loan
job_self_employed
job_technician
housing
job_housemaid
age
cons_price_idx
job_services
job_blue_collar
contact_count
contact_month_aug
prev_contact
contact_month_nov
contact_month_may
n_employed
0.30
0.15
0.10
0.07
0.07
0.06
0.06
0.05
0.04
0.04
0.04
0.04
0.03
0.03
0.02
0.01
0.00
-0.00
-0.00
-0.00
-0.00
-0.01
-0.01
-0.01
-0.01
-0.01
-0.01
-0.01
-0.01
-0.02
-0.02
-0.02
-0.02
-0.04
-0.06
-0.06
-0.08
-0.09
-0.29
-0.62
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
Coefficient

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

There are too many low coefficients for an easily readable graph, so I slice out the larger ones and plot them.

[185]:
```
# Slice out only the large coefficients;
major_coef = coef.loc[coef.abs() >= .1].copy()

# Plot heated bars
ax = plotting.heated_barplot(data=major_coef,
                             figsize=(5, 5))

# Add annotations
plotting.annot_bars(ax, dist=.25)
ax.set_xlim(-1.1, .6)

ax.set(title="Final Model Coefficients", xlabel="Coefficient",);
```




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

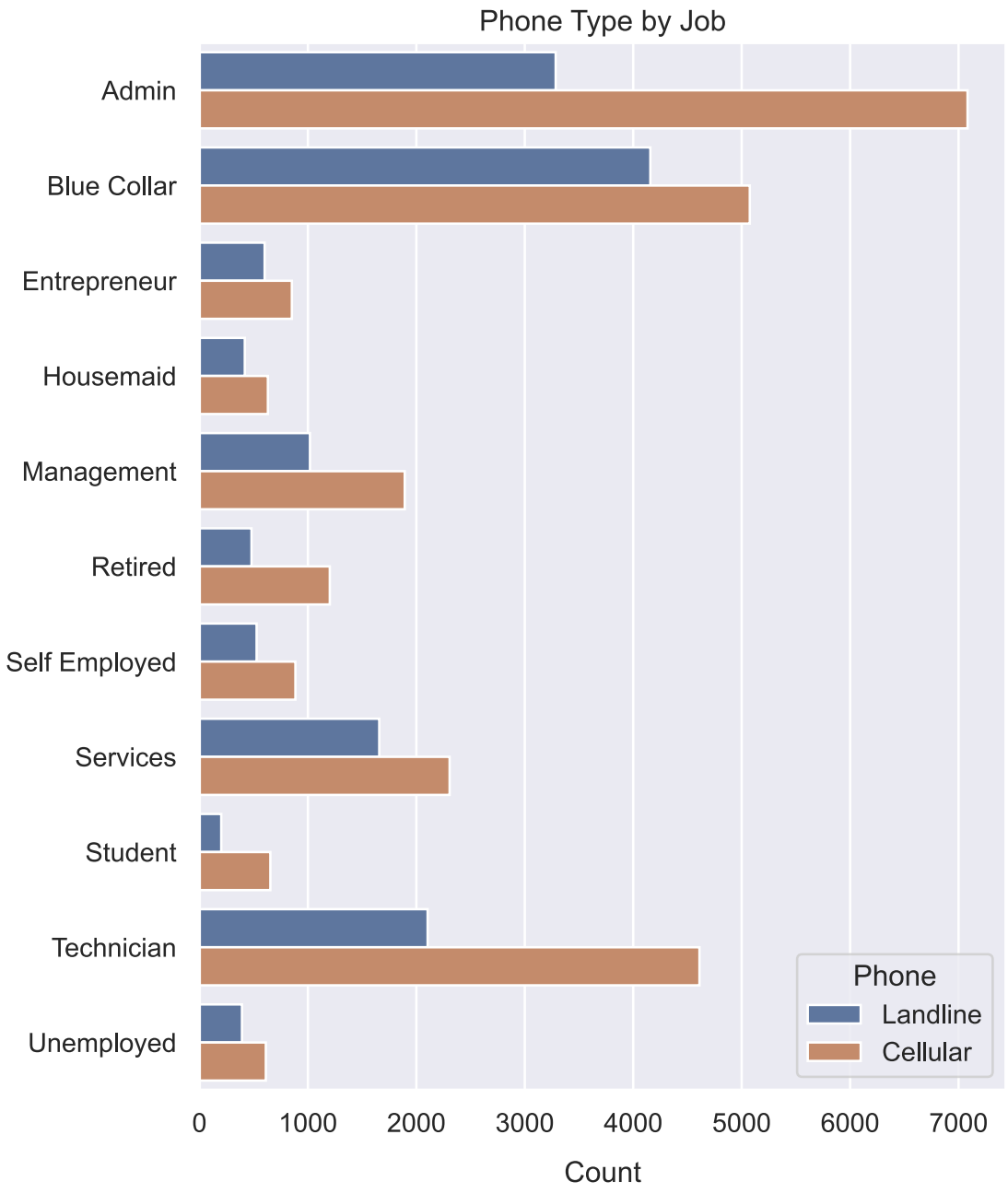
7.1 Positive Coefficients

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.

[186]:
```
# Create special title-formatted DataFrame
title_df = utils.title_mode(df)
title_df["Phone"] = title_df["Contact Cellular"].map(
    lambda x: "Cellular" if x else "Landline")

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

[186]:
```
[Text(0, 0.5, ''), Text(0.5, 0, 'Count'), Text(0.5, 1.0, 'Phone Type by Job')]
```

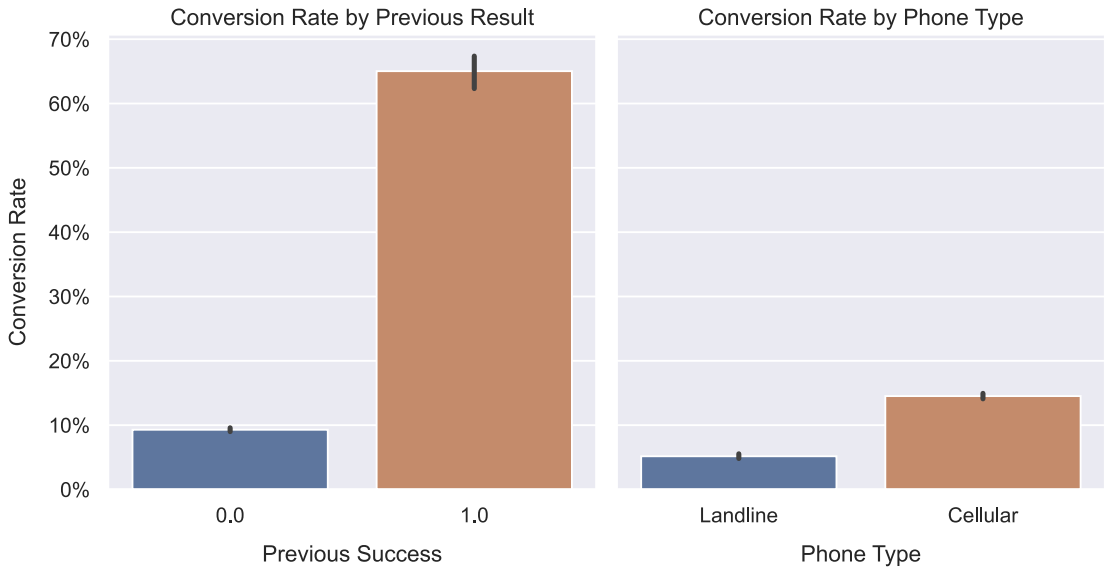



```
<IPython.core.display.Javascript object>
```

[187]:
```
# Plot conversion rates for positive coeff features
fig = plotting.multi_rel(data=title_df,
                         x=["Prev Success", "Phone"],
                         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",
           title="Conversion Rate by Previous Result",
           xlabel="Previous Success")
axs[1].set(title="Conversion Rate by Phone Type",
           xlabel="Phone Type")
```

[187]:
```
[Text(0.5, 1.0, 'Conversion Rate by Phone Type'),
 Text(0.5, 1812.9646808510638, 'Phone Type')]
```

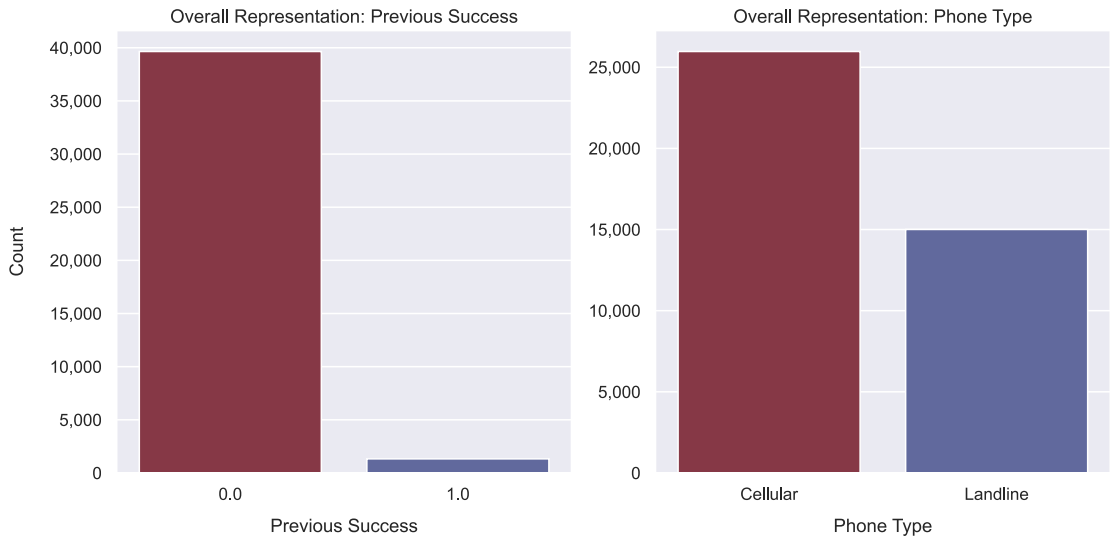



```
<IPython.core.display.Javascript object>
```

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?

It is certainly more popular.

[188]:
```
fig = plotting.multi_countplot(data=title_df[["Prev Success", "Phone"]],␣
↪orient="v", annot=False);
axs = fig.get_axes()
axs[0].set(xlabel="Previous Success", title="Overall Representation: Previous␣
↪Success")
axs[1].set(xlabel="Phone Type", title="Overall Representation: Phone Type")
del title_df
```




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

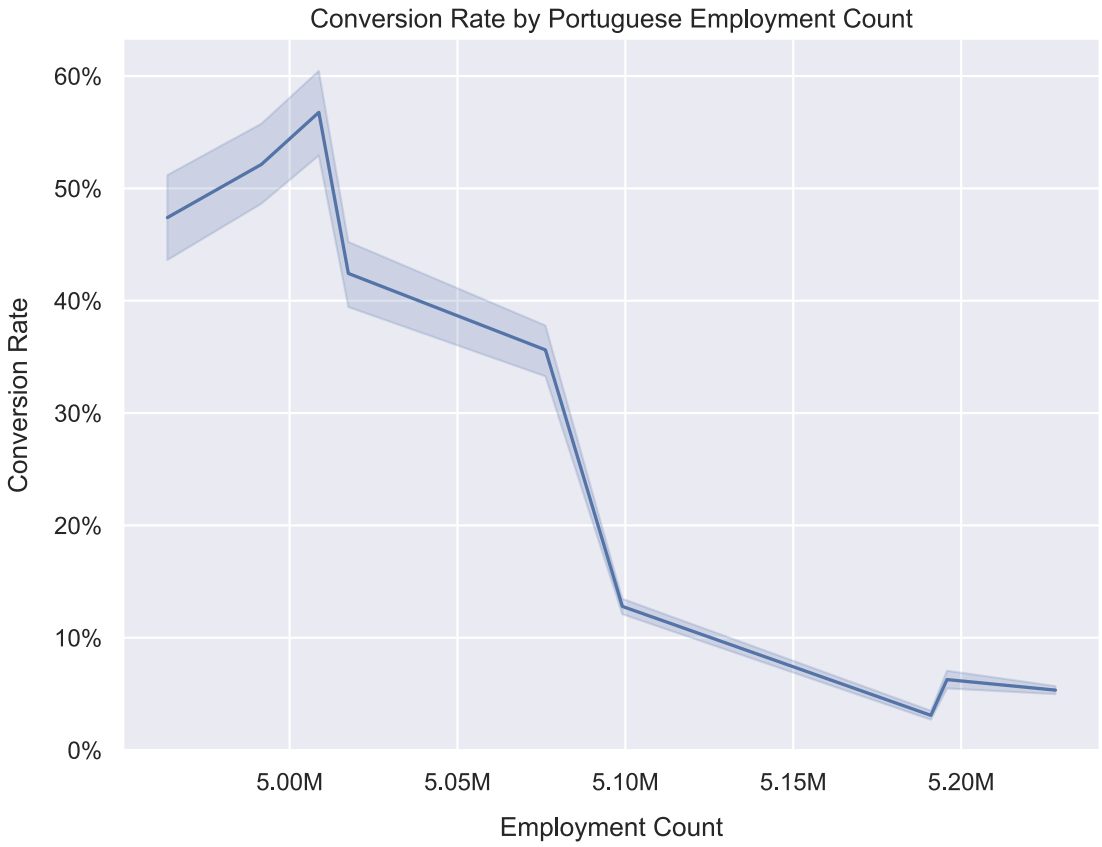
7.2 Negative Coefficients

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.

[189]:
```
# 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")
```

[189]:
```
[Text(0, 0.5, 'Conversion Rate'),
 Text(0.5, 0, 'Employment Count'),
 Text(0.5, 1.0, 'Conversion Rate by Portuguese Employment Count')]
```




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

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.

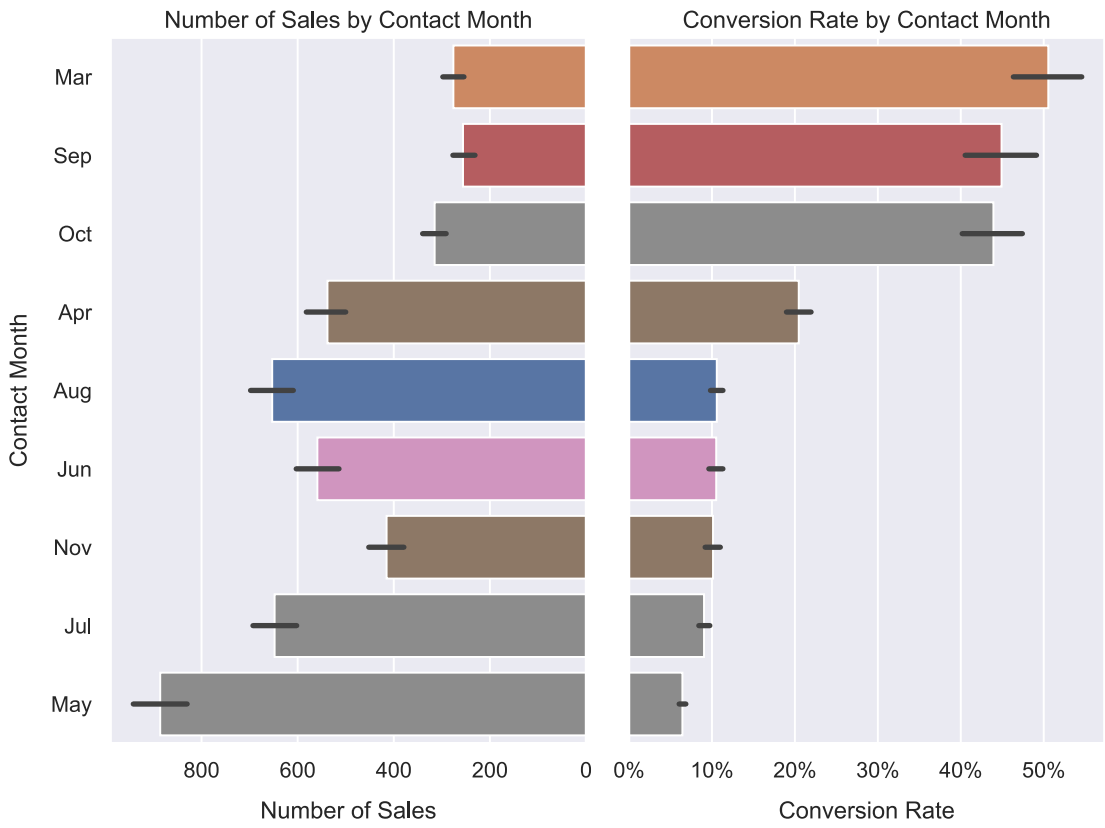
[190]:
```
df.query("job == 'unemployed'")["invested"].value_counts(1)
```

[190]:
```
0.0    0.861554
1.0    0.138446
Name: invested, dtype: float64
```

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

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.

[191]:
```
conversion_bars(data=df, y="contact_month", size=(4, 6));
```



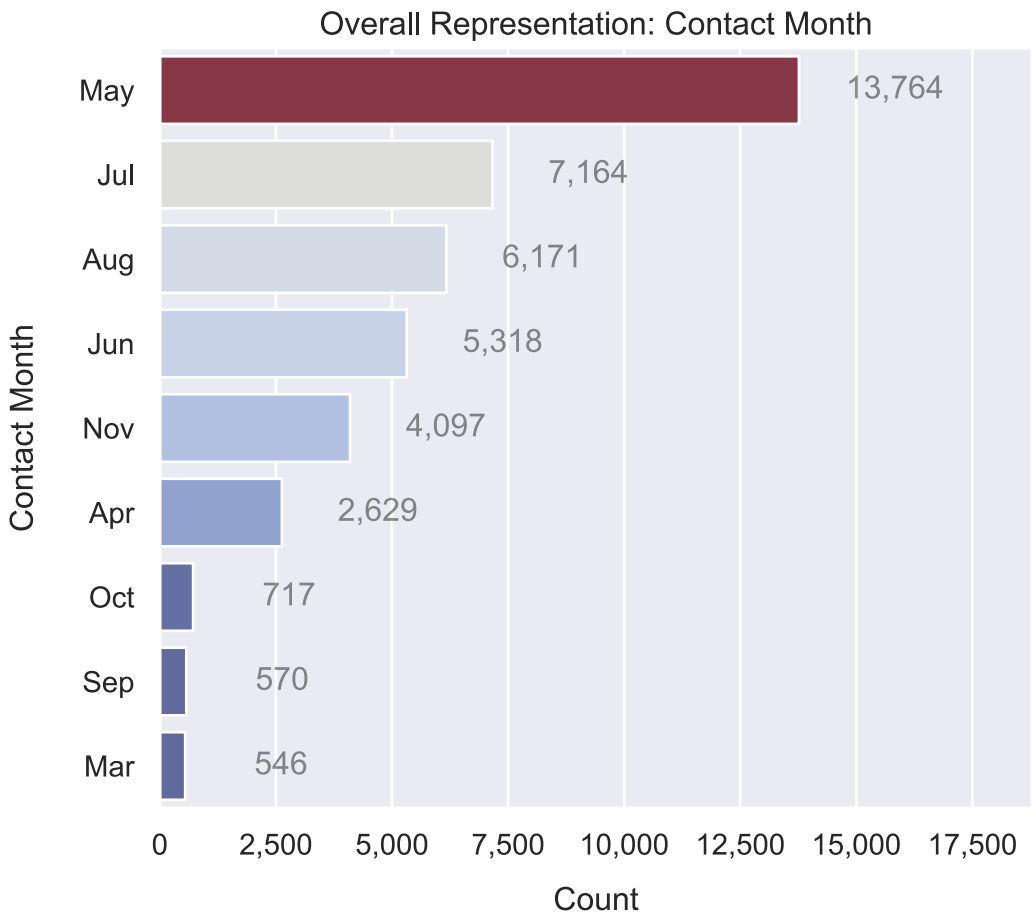

```
<IPython.core.display.Javascript object>
```

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.

[192]:
```
fig = plotting.multi_countplot(data=df[["contact_month"]]);
axs = fig.get_axes()
axs[0].set(title="Overall Representation: Contact Month")
plotting.map_ticklabels(axs[0], utils.to_title, axis="y")
```




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

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. #### Relax more than you're used to in Summer months. May is a strikingly bad month for conversion efficiency. Evidently much effort is expended in the summer, but it's not very efficient. #### 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.

Another unrelated idea is to try testing the models using a temporal train-test split rather than a random split. For example, the earliest 75% of observations could serve as the training set and the latest 25% could serve as the test set. This would be an even more realistic way to test a model's predictive capabilities. While I do not have access to the date of each observation, I do know that they are ordered by date. Since I do have access to the month of each observation, I could also potentially infer the dates.

I would also like to try using sophisticated imputation techniques to fill some of the large missing value gaps in the dataset. I would like to experiment with Scikit-Learn's `IterativeImputer` and `KNNImputer` to see if advanced imputation leads to better predictions than simply filling nulls with 0. Since there are enormous gaps for features like 'prev_outcome' (i.e. the vast majority are missing), I'm curious whether advanced imputation techniques make things better or worse.

-