Solution - Bank Direct Marketing

Solution: Bank Direct Marketing

Import Libraries and Dataset

In [1]:
```
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set_style('darkgrid')
```

In [2]:
```
df = pd.read_csv('data/bank_direct_marketing.csv')
```

Exploratory Analysis and Data Cleaning

We'll start by "getting to know" the dataset. The goal is to understand the dataset at a qualitative level, note anything that should be cleaned up, and spot opportunities for feature engineering.

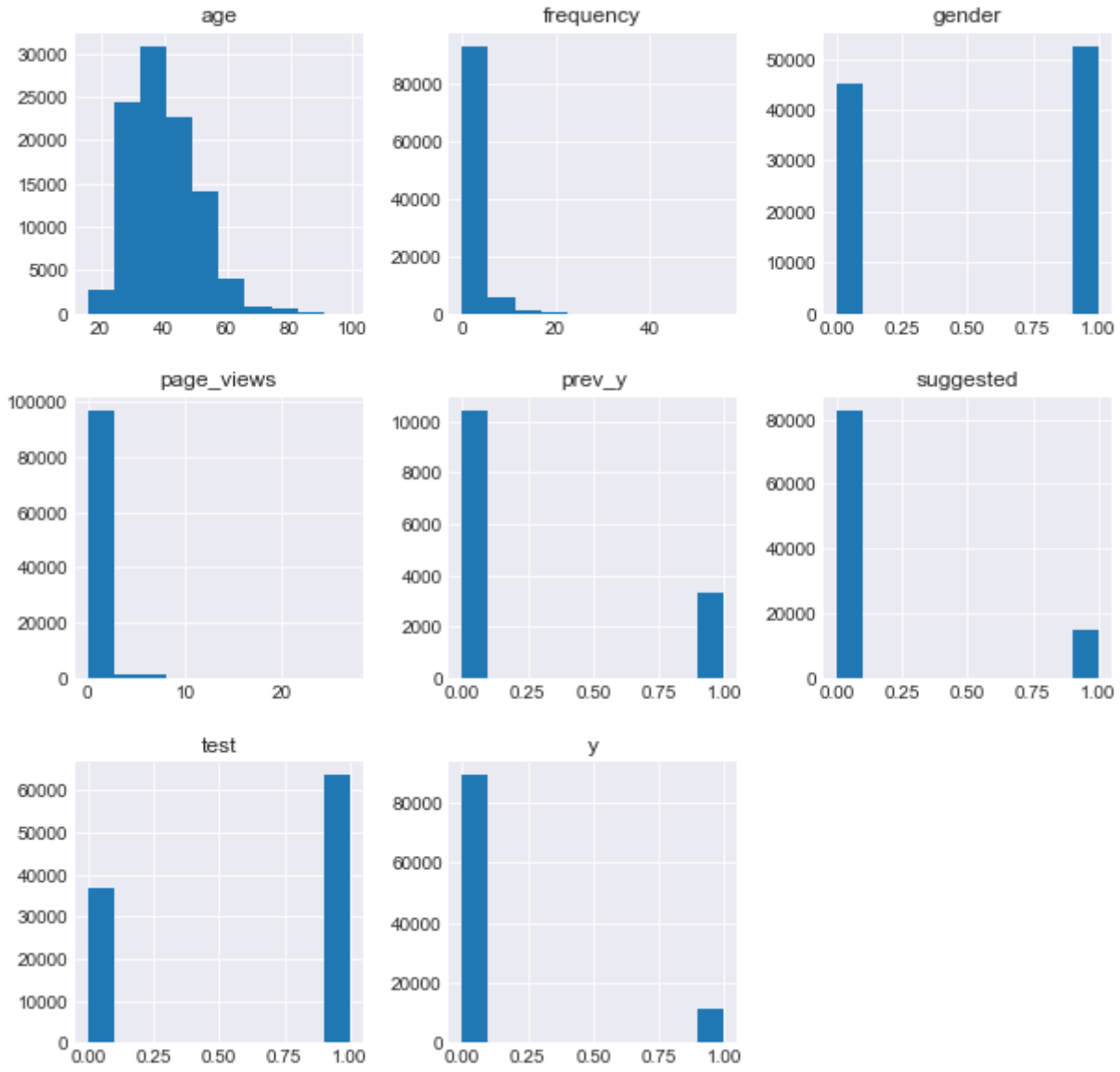
Tip: In our Machine Learning Masterclass, we separate exploratory analysis and data cleaning into separate steps. However, because take-home challenges are meant to be shorter/condensed analyses, feel free to combine these steps. Just remember to keep your code clean and organized.

In [3]:
```
# Example observations
df.head()
```

Out[3]:

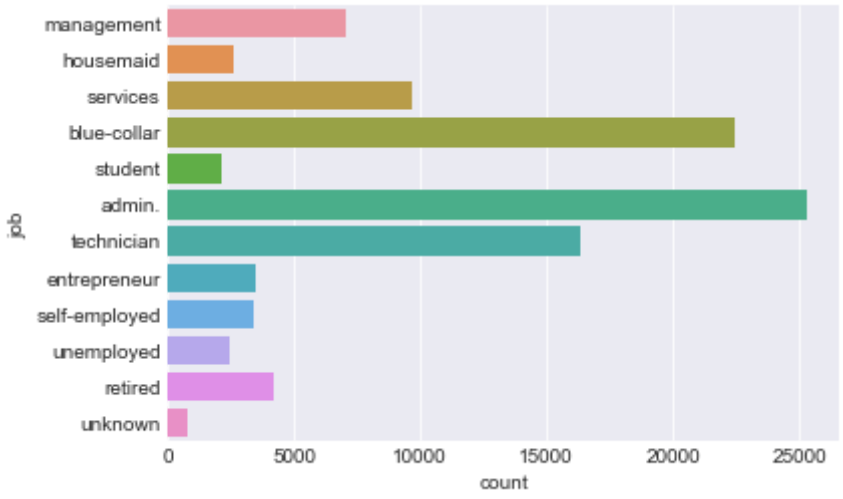
	age	job	marital	education	gender	suggested	test	day_of_week	frequency	page_views
0	43	management	married	high.school	0.0	0.0	1	wed	3	0
1	34	housemaid	married	basic.4y	1.0	0.0	1	mon	2	0
2	27	services	single	basic.9y	1.0	0.0	1	tue	1	0
3	52	services	married	high.school	0.0	0.0	1	tue	0	0
4	31	blue-collar	married	basic.9y	1.0	0.0	1	thu	1	0

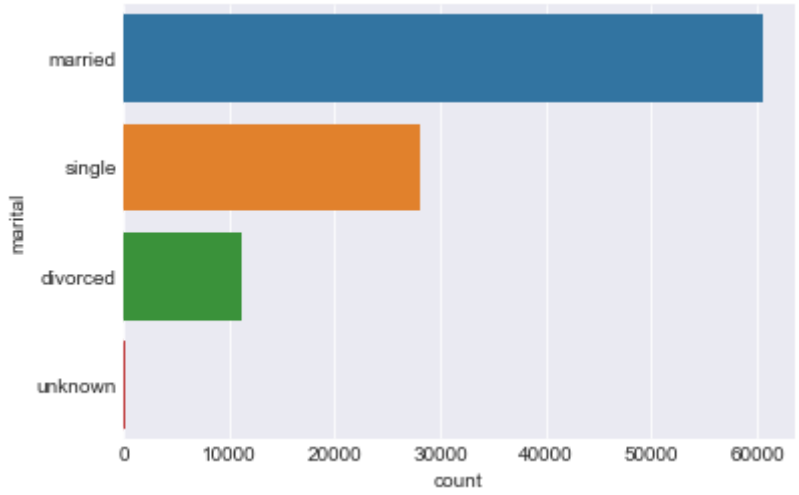
In [4]:
```
ax = df.hist(figsize=(10,10))
```

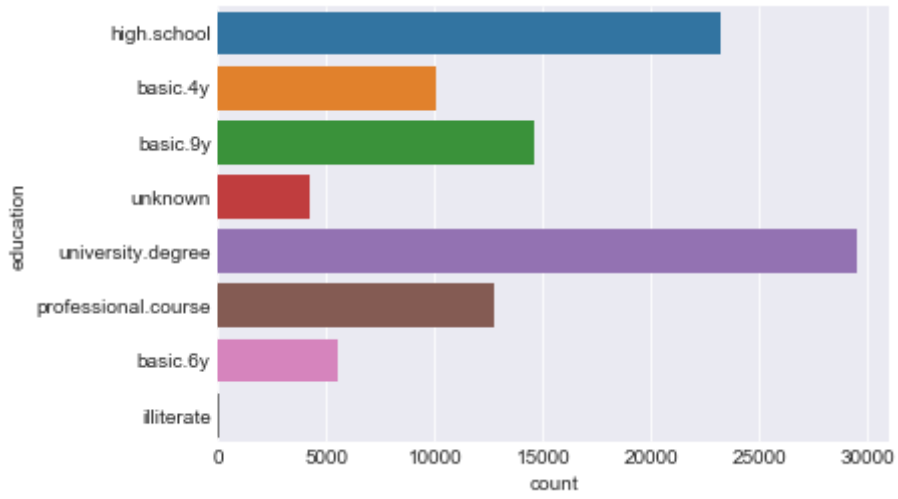


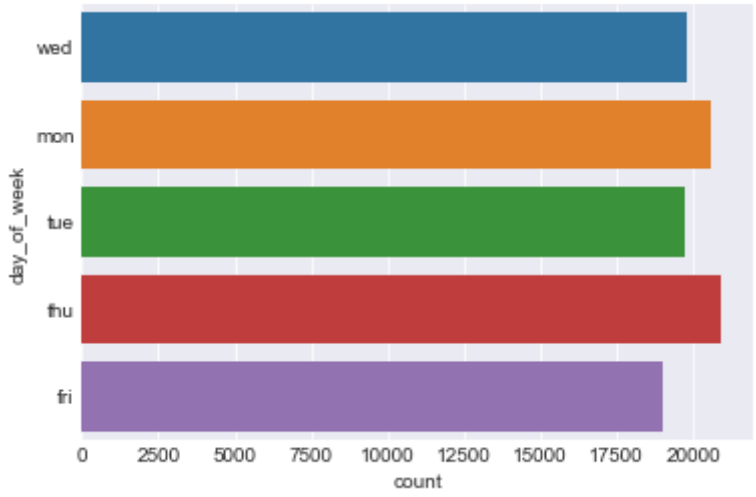
Distributions of categorical features:

In [5]:
```
sns.countplot(y="job", data=df)
plt.show()

sns.countplot(y="marital", data=df)
plt.show()

sns.countplot(y="education", data=df)
plt.show()

sns.countplot(y="day_of_week", data=df)
plt.show()
```


management
housemaid
services
blue-collar
student
admin.
technician
entrepreneur
self-employed
unemployed
retired
unknown
job
0
5000
10000
15000
20000
25000
count


married
single
divorced
unknown
marital
0
10000
20000
30000
40000
50000
60000
count


high.school
basic.4y
basic.9y
unknown
university.degree
professional.course
basic.6y
illiterate
education
0
5000
10000
15000
20000
25000
30000
count



Check for missing values:

In [6]:
```
df.isnull().sum()
```

Out[6]:
```
age                0
job                0
marital            0
education          0
gender          2405
suggested       2405
test               0
day_of_week        0
frequency          0
page_views         0
prev_y         86264
y                  0
dtype: int64
```

Flag and fill missing values.

In [7]:
```
# Flag missing gender and suggested values
df['gender_missing'] = df.gender.isnull() * 1
df['suggested_missing'] = df.suggested.isnull() * 1

# Recipients who have not received previous campaign
df['prev_NA'] = df.prev_y.isnull() * 1

# Fill missing values
df.fillna(0, inplace=True)
```

In [8]:
```
df.isnull().sum()
```

Out[8]:
```
age                0
```

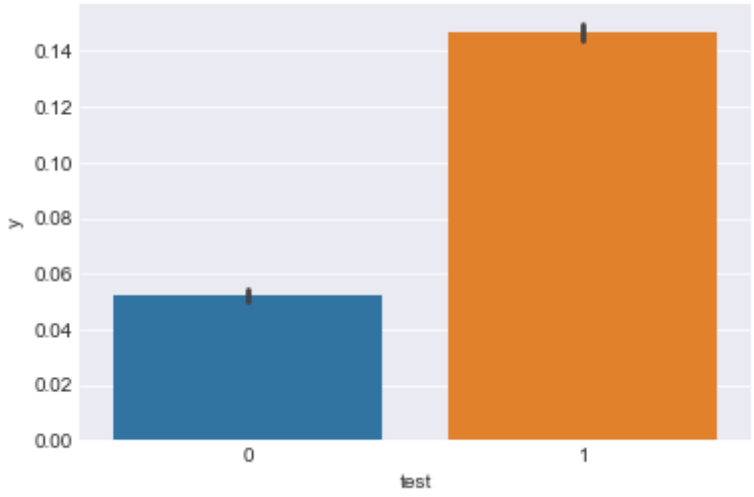
```
job                 0
marital             0
education           0
gender              0
suggested           0
test                0
day_of_week         0
frequency           0
page_views          0
prev_y              0
y                   0
gender_missing      0
suggested_missing   0
prev_NA             0
dtype: int64
```

A/B Test Results

Analyzing A/B test results is not difficult. There are different approaches, including calculating significance with statistical t-tests, but you can often just look at the plots and 95% confidence interval bars:

In [9]:
```
sns.barplot(x='test', y='y', data=df, ci=95)
```

Out[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1483bcc0>
```

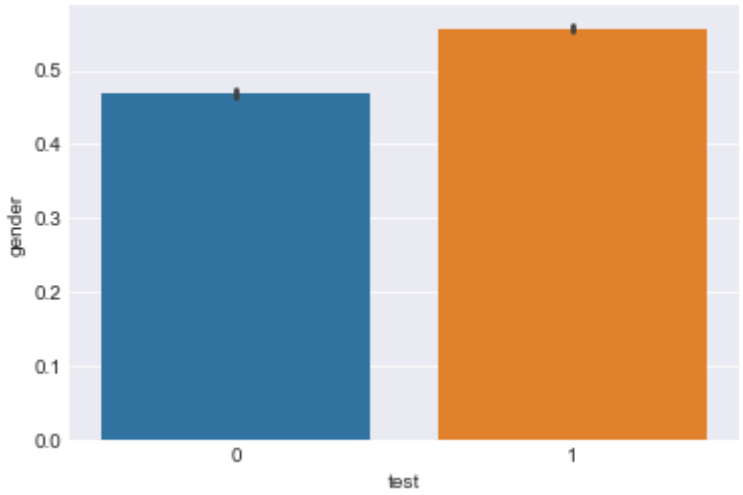


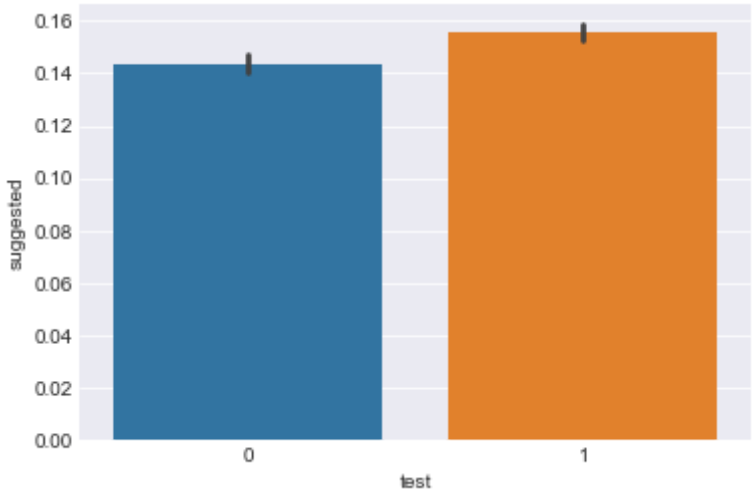
This is a case where the test conversion rate clearly beats the control.

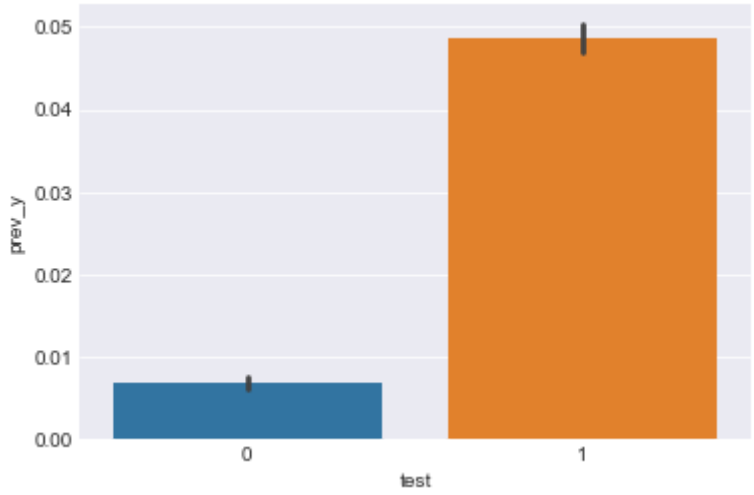
However, before we write the test as a huge success, we should check a few things, especially because we did not design the test ourselves (i.e. we were just given the dataset from the bank).

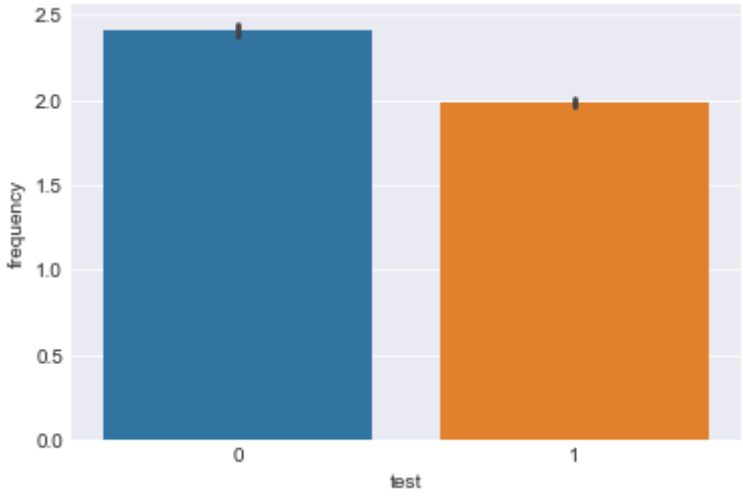
Specifically, we should check if the test recipients and control recipients came from the same population - whether they have similar distributions along key features.

In [10]:
```
# Compare test vs. control gender
sns.barplot(x='test', y='gender', data=df)
plt.show()

# Compare test vs. control suggested
sns.barplot(x='test', y='suggested', data=df)
plt.show()

# Compare test vs. control past campaign
sns.barplot(x='test', y='prev_y', data=df)
plt.show()

# Compare test vs. control frequency
sns.barplot(x='test', y='frequency', data=df)
plt.show()
```




0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
suggested
0
1
test


0.05
0.04
0.03
0.02
0.01
0.00
prev_y
0
1
test



It's very clear that our test group and control group were **not** randomly selected.

Yet this is often the reality in business - often, we cannot have perfect testing environments, and data scientists are frequently tasked with the (theoretically impossible) mission of extracting insights from poorly designed tests.

So what do we do?

Well, one approach is to build out a full model and then see if the `test` feature is still important, then plot the partial dependences. Let's proceed with this approach.

Machine Learning

Note: Remember, for take-home challenges, you don't need to spend an immense amount of time optimizing a model. Just explain your choice of algorithm and mention how you would tune it to improve performance if given more time.

For this challenge, we'll use a Gradient Boosted Tree. In practice, Gradient Boosted Trees often see the best performance after properly tuning them, and they tend to be more robust to outliers. In addition, we can use sklearn's `plot_partial_dependence()` function.

Partial dependence plots show the dependence between a function (i.e. our classifier) and a set of features, marginalizing over the values of all other features (the complement features). We can use this to understand the effect of our test while accounting for other features.

In [11]:
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble.partial_dependence import plot_partial_dependence
```

First, we'll create the analytical base table.

In [12]:
```
abt = pd.get_dummies(df)
```

Then, we'll split training and test sets.

In [13]:
```
y = abt.y
X = abt.drop('y', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=11
```

Next, we'll train a basic Gradient Boosted Tree. Again, we won't spend much time optimizing it for the take-home challenge, but make a note of the hyperparameters you would tune if given more time. In this case, the three most impactful hyperparameters for Gradient Boosted Trees are:

1. Number of trees
2. Learning rate
3. Max depth of each tree

We would pass a hyperparameter grid into a `GridSearchCV` object to tune.

But for now, we'll train a "default" classifier:

In [14]:
```
clf = GradientBoostingClassifier(random_state=123)

clf.fit(X_train, y_train)

pred = clf.predict_proba(X_test)
pred = [p[1] for p in pred]
```

Next, we'll see if the model has reasonable performance.

In [15]:
```
roc_auc_score(y_test, pred)
```

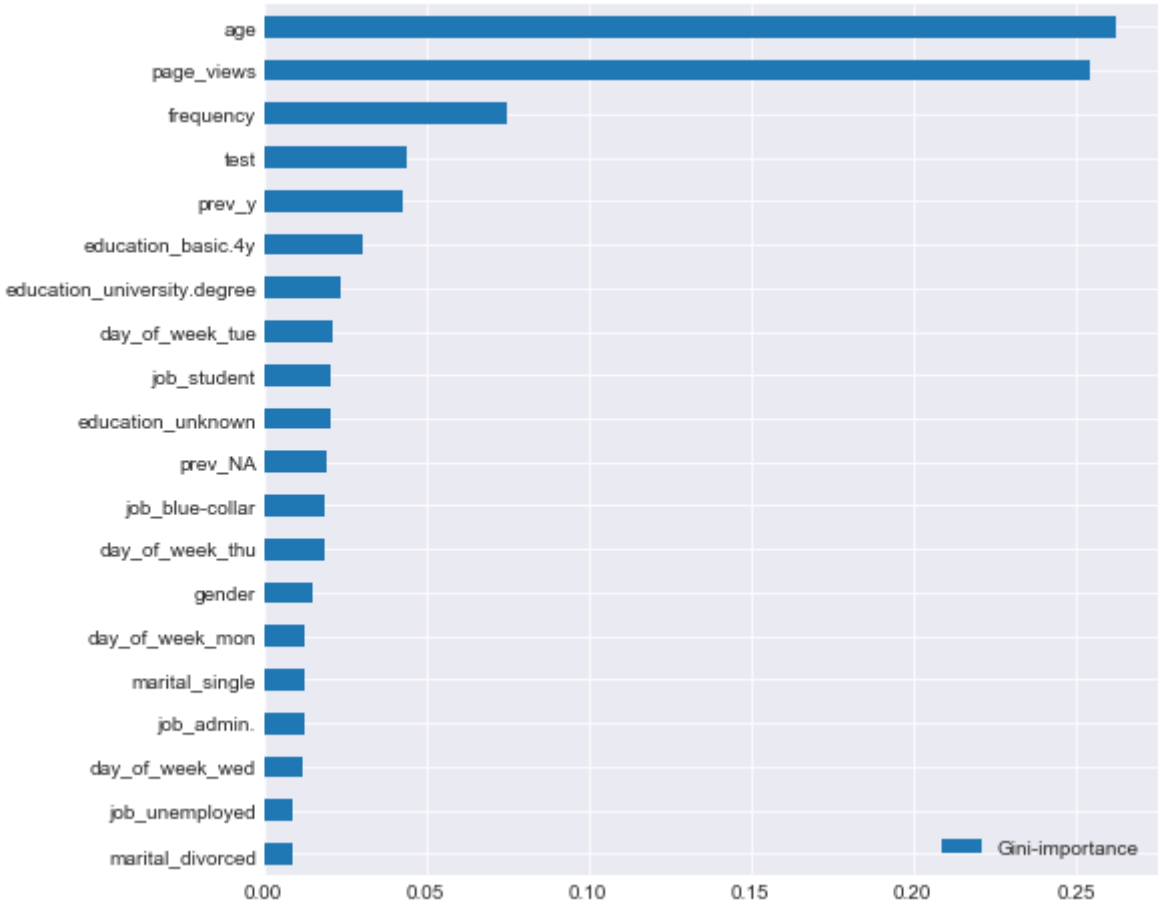
Out[15]:
```
0.7348790426052461
```

An AUROC score of 0.73 right out of the gate is not bad. It means that if we are randomly given one observation for `y==1` and one for `y==0`, we'd be able to distinguish them 73% of the time. The model can surely be improved, but this is a decent enough benchmark for us to continue with the analysis.

Next, we'll plot feature importances to see if `test` is still important after accounting for other factors.

In [16]:
```
# Helper function for plotting feature importances
def plot_feature_importances(columns, feature_importances, show_top_n=10)
    feats = dict( zip(columns, feature_importances) )
    imp = pd.DataFrame.from_dict(feats, orient='index').rename(columns={(
    imp.sort_values(by='Gini-importance').tail(show_top_n).plot(kind='bar
    plt.show()
```

In [17]:
```
plot_feature_importances(X_train.columns, clf.feature_importances_, 20)
```



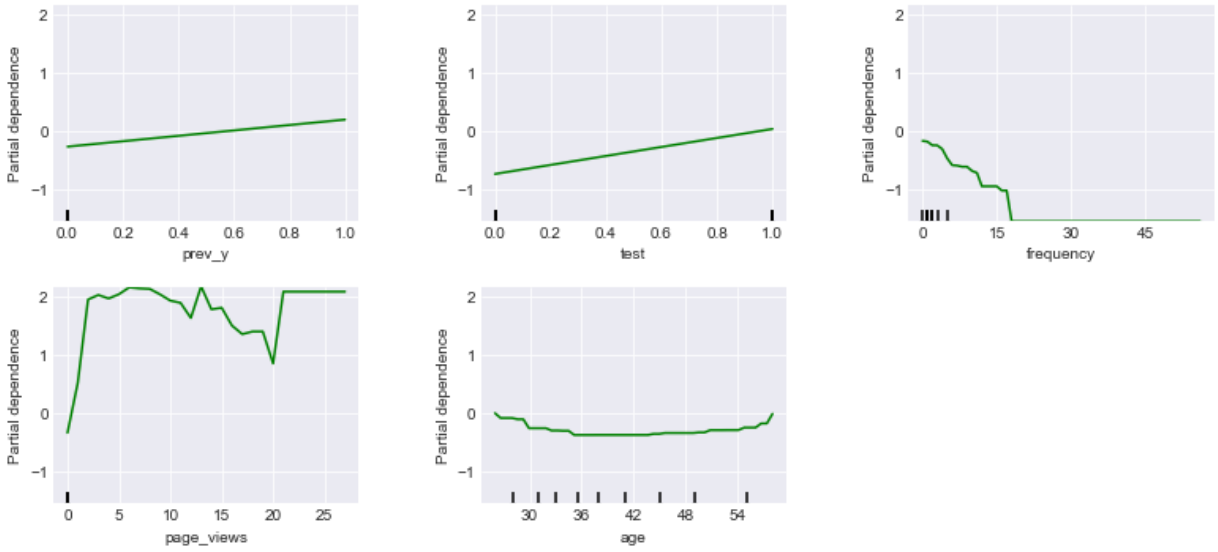
As you can see, `test` makes it into the top 5 most important features. Let's plot partial dependence plots for those top 5. (You can read more about partial dependence plots here.)

In [18]:
```
fi = clf.feature_importances_

# Get positions of top 5 features
top5_features = sorted(range(len(fi)), key=lambda i: fi[i])[-5:]
```

In [19]:
```
# Check that we have the right features
X_train.columns[top5_features]
```

Out[19]:
```
Index(['prev_y', 'test', 'frequency', 'page_views', 'age'], dtype='objec
```

In [20]:
```
# Partial depedence plots
plot_partial_dependence(clf,
                        X_train,
                        top5_features,
                        feature_names=X_train.columns,
                        n_jobs=3,
                        grid_resolution=50,
                        figsize=(12,8))
plt.show()
```



As you can see, even after accounting for the other 4 features, `test` still has the positive impact on response rate.

All in all, despite the suspect A/B test setup, we can reasonably believe that the test had a positive affect on response rates.