In [1]:

```
import pandas as pd
import numpy as np

# plotting
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# preprocessing
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, RobustScaler
from sklearn.compose import ColumnTransformer
from dirty_cat import TargetEncoder
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.feature_selection import SelectFromModel

# modeling
from sklearn.linear_model import LogisticRegression, RidgeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.tree import DecisionTreeClassifier, export_graphviz

# visualizing trees
import graphviz

# model evaluation
from sklearn.metrics import precision_recall_curve, roc_curve

# warnings
import warnings
warnings.filterwarnings('ignore')

# custom functions
from hw3_functions import *

# other
pd.set_option('display.max_columns', 100)
np.random.seed(42)
```

# HW3 - Payment Classification

By Corentin Llorca (cl3783) and Costas Vafeades (cv2451)

Goal is to predict whether a payment by a company to a medical doctor or facility was made as part of a research project or not. All relevant data can be found here (https://www.cms.gov/OpenPayments/Explore-the-Data/Dataset-Downloads.html).

### Data Description

Physicians may be identified as covered recipients of records or as principal investigators associated with research-related payment records. Teaching hospitals may also be identified as covered recipients. Teaching hospitals are defined as any hospital receiving payments for GM, IPPS or IME.

Each record in the General Payment, Research Payment, and Ownership/Investment files includes a Change Type indicator field.

- NEW: the record is newly reported by the reporting entity since the last publication and is being published for the first time.
- ADD: the record is not new in the system but, due to the record not being eligible for publication until the current publication cycle, is being published for the first time.
- CHANGED: record was previously published but has been modified since its last publication. A record whose only change since the last publication is a change to its dispute status is categorized as a changed record.
- UNCHANGED: record was published during the last publication cycle and is being republished without change in the current publication.

## Task 1: Identify Features

First of all, let's load the data to assemble the dataset. The data comes from two different csv files, OP_DTL_GNRL_PGYR2017_P01182019.csv (general payments) and OP_DTL_RSRCH_PGYR2017_P01182019.csv (research payments), so we load a subsample of those two files, add the target feature "research_payment" (0 for rows in the first file and 1 for rows in the second file) and concatenate them.

### How do we balance classes?

The data is naturally imbalanced, as there are much more records of general payments than research payments. Here are the row counts for both files:

In [2]:

```
n_gen = sum(1 for line in open('data\OP_DTL_GNRL_PGYR2017_P01182019.csv')) - 1
n_res = sum(1 for line in open('data\OP_DTL_RSRCH_PGYR2017_P01182019.csv')) - 1
# the -1 is to exclude header

print("General payments: " + str(n_gen) + " lines")

print("Research payments: " + str(n_res) + " lines")
```

```
General payments: 10663833 lines
Research payments: 602530 lines
```

Since the general payments csv is way too large to load, we will select a subsample of the rows for each csv, forming the whole dataset. We have the choice of how many rows to select in each file, and the choice we make will end up deciding the class balance. Here are the two options we have:

- Select an equal number of rows for both classes: this completely removes class imbalance and the problems it might cause. However, we lose the "real-world setting" with imbalanced classes
- Select a number of rows in each file that's proportionate to their total number of rows: this will cause class imbalance problems since there is an approximate 95% / 5% class distribution, but will reflect the whole problem better.

We ended up choosing the former.

## Loading and joining the datasets

We first load the separate datasets and add the target feature. To load the data, we do a random subsampling.

In [3]:

```
# Number of desired samples for each file
nsamples_gen = 20000
nsamples_res = 20000

skiprows_gen = np.sort(np.random.choice(range(1, n_gen+1), replace = False, size = n_ge
n - nsamples_res))
skiprows_res = np.sort(np.random.choice(range(1, n_res+1), replace = False, size = n_re
s - nsamples_res))

gen = pd.read_csv("data\OP_DTL_GNRL_PGYR2017_P01182019.csv", skiprows = skiprows_gen, p
arse_dates=['Date_of_Payment'])
res = pd.read_csv("data\OP_DTL_RSRCH_PGYR2017_P01182019.csv", skiprows = skiprows_res,
parse_dates=['Date_of_Payment'])

gen['research_payment'] = 0
res['research_payment'] = 1
```

### Concatenation

The next problem is to concatenate the data. This raises an issue: our two data files have different columns - but also have a lot of columns in common. The "baseline" choice here would be to simply use panda's concatenate function, which would give us a concatenated dataset, whose columns would be the union of the columns of the two separated datasets, filling the missing values with NA.

However, this creates a problem: since the two separated datasets are also the two separated classes, then if one feature is only non-missing in one of the classes, it might indirectly reveal information about the class to the model, in an unwanted way (data leakage). The solution to that is to only keep the features that are in both datasets when concatenating (inner join instead of outer join).

In [4]:

```
df = pd.concat([gen, res], join='inner')
df = df.reset_index(drop=True)
```

## Data Types

We need to check that pandas has loaded the right data types for our dataframe. Printing the dtypes shows us a few irregularities, namely that there are ID features that are counted as floats/ints when they should be used as categoricals. Other than that, there seems to be no problem. We notice, by looking at the dtypes, that the whole dataset has only one continuous variable (Total_Amount_of_Payment_USDollars).

In [5]:

```
to_cat = ['Teaching_Hospital_CCN', 'Teaching_Hospital_ID',
          'Physician_Profile_ID', 'Physician_License_State_code5',
          'Record_ID', 'Program_Year', 'Applicable_Manufacturer_or_Applicable_GPO_Making_Payment_ID']

df = df.astype(dtype={v: object for v in to_cat}, copy=False)
```

## Checking for Data Leakage and Irrelevant Features

Before we proceed, it's important to check the number of unique values of each of our variables to make sure that they don't leak the target and also remove irrelevant features.

In [6]:

```
df.groupby('research_payment').nunique()
```

Out[6]:

| research_payment | Change_Type | Covered_Recipient_Type | Teaching_Hospital_CCN | Teachi |
|---|---|---|---|---|
| **0** | 3 | 2 | 67 | |
| **1** | 3 | 4 | 1772 | |

We can remove Record_ID as that's a unique identifier of each observation.

In [7]:

```
cols_drop = ['Record_ID']
```

Variables that have the same unique value for both research and non-research payments can be considered as irrelevant and be dropped.

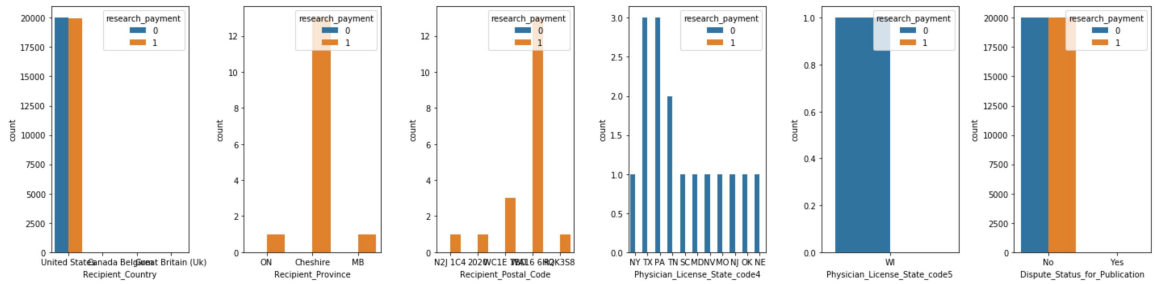
In [8]:

```
for v in df.columns.values[:-1]:
    if len(df[v].unique()) == 1:
        cols_drop.append(v)

df.drop(cols_drop, axis=1, inplace=True)
```

Variables that have only one value for either of the two types of payments should be investigated further.

In [9]:

```
cols_drop = []
for v in df.columns.values[:-1]:
    for i in range(2):
        if len(df[df['research_payment'] == i][v].unique()) == 1:
            if v not in cols_drop: cols_drop.append(v)


fig, ax = plt.subplots(1, len(cols_drop), figsize = (20, 5))
i=0
for v in cols_drop:
    sns.countplot(v, hue='research_payment', data = df, ax = ax[i])
    i+=1

plt.tight_layout()
```




We end up also dropping all of those, since they really don't seem to contain information that's not data leakage.

In [10]:

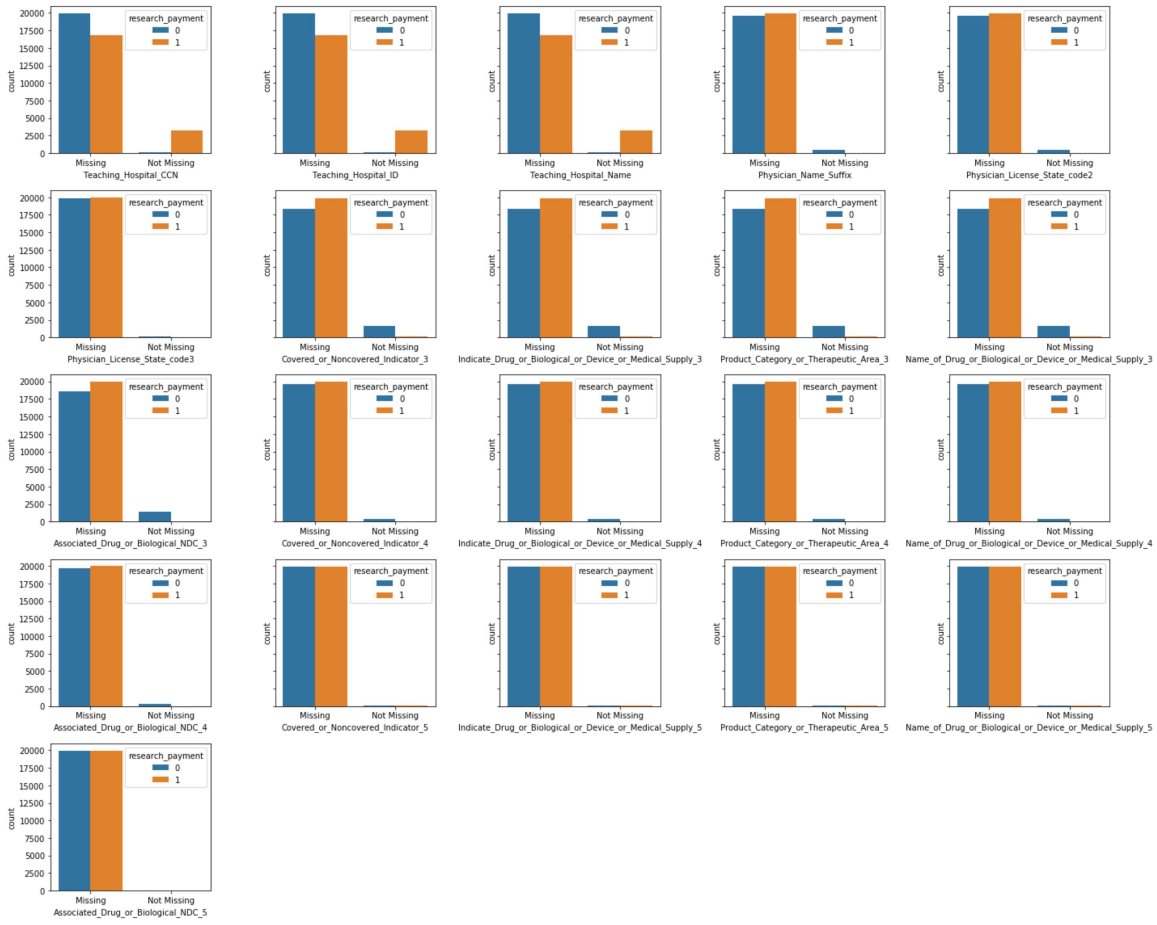
```
df.drop(cols_drop, axis=1, inplace=True)
```

### Sparse Variables

A big portion of our features don't have any data for one of the targets.

In [11]:

```
df_missing = df.copy()
for c in df_missing.columns[:-1]:
    df_missing[c] = np.where(df_missing[c].isnull(), 'Missing', 'Not Missing')

fig, ax = plt.subplots(5, 5, figsize = (20, 16), sharey='row')
cols_drop = []
# axis indicators
i = 0
j = 0

for v in df_missing.columns.values[:-1]:
    if df_missing[df_missing[v] == 'Missing'].shape[0] > (df_missing.shape[0] * 0.9):
        sns.countplot(v, hue='research_payment', data = df_missing, ax=ax[j, i])
        cols_drop.append(v)
        # increment axis
        i += 1
        if i == 5: j += 1; i = 0


fig.delaxes(ax[4, 1])
fig.delaxes(ax[4, 2])
fig.delaxes(ax[4, 3])
fig.delaxes(ax[4, 4])
plt.tight_layout()
plt.show()
```



Those variables contain very little information due to the amount of missing values, and some of them have non-missing values for only one of the two classes, which also indicates data leakage. We decided to drop all of these variables too.

In [12]:

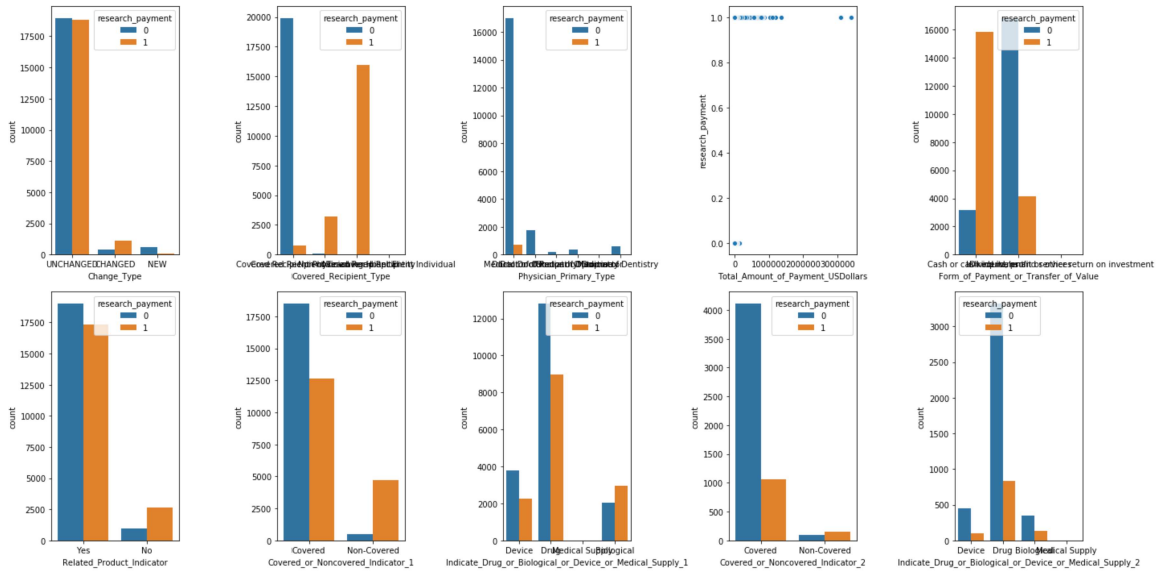
```
df.drop(cols_drop, axis=1, inplace=True)
```

### Leaky Variables

We're now plotting the variables that we haven't dropped yet and that don't have too many categories (in order to make the plot readable). We hope to observe cases of data leakage by doing that.

In [13]:

```
fig, ax = plt.subplots(2, 5, figsize=(20, 10))
i = 0
j = 0

obs_vars = []

for c in df.columns.values[:-1]:
    if df[c].dtype == 'float64':
        sns.scatterplot(x=c, y='research_payment', data=df, ax=ax[j, i])
        i+=1
        obs_vars.append(c)
        if i == 5: j+=1; i=0
    elif (len(df[c].unique()) <= 10) & (df[c].dtype == object):
        sns.countplot(x=c, hue='research_payment', data=df, ax=ax[j, i])
        i+=1
        obs_vars.append(c)
        if i == 5: j+=1; i=0


plt.tight_layout()
plt.show()
```



Judging by the plots above, the variables Covered_Recipient_Type, Form_of_payment and all the Physician variables could be leaking the target. It's important to investigate these further before proceeding.

**Covered_Recipient_Type**

In [14]:

```
df.groupby(['research_payment', 'Covered_Recipient_Type']).size().reset_index()
```

Out[14]:

| | research_payment | Covered_Recipient_Type | 0 |
|---|---|---|---|
| **0** | 0 | Covered Recipient Physician | 19933 |
| **1** | 0 | Covered Recipient Teaching Hospital | 67 |
| **2** | 1 | Covered Recipient Physician | 778 |
| **3** | 1 | Covered Recipient Teaching Hospital | 3228 |
| **4** | 1 | Non-covered Recipient Entity | 15976 |
| **5** | 1 | Non-covered Recipient Individual | 18 |

Given that the proportions are very different for each of our targets, this raises further suspicions. General payments are only made to covered recipients so we can remove this variable as it's better to be safe than sorry.

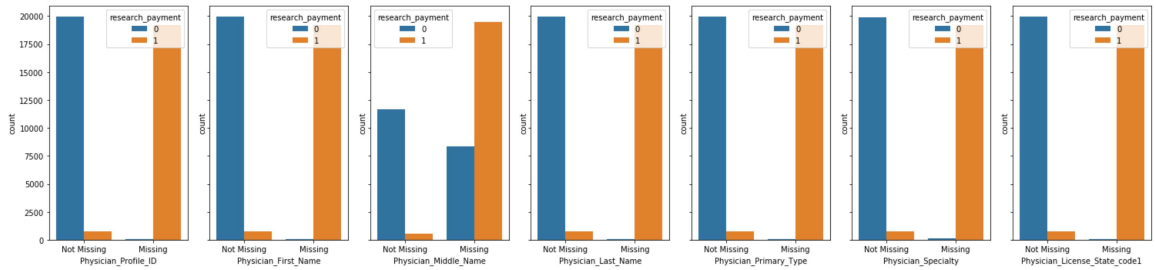
In [15]:

```
df.drop('Covered_Recipient_Type', axis=1, inplace=True)
```

**Physician Variables**

In [16]:

```
physician_vars = ['Physician_Profile_ID', 'Physician_First_Name', 'Physician_Middle_Nam
e', 'Physician_Last_Name',
                  'Physician_Primary_Type', 'Physician_Specialty', 'Physician_License_S
tate_code1']

fig, ax = plt.subplots(1, 7, figsize=(21, 5), sharey='row')
i = 0

for v in physician_vars:
    sns.countplot(np.where(df[v].notnull(), 'Not Missing', 'Missing'), hue=df['research
_payment'], ax=ax[i])
    ax[i].set_xlabel(v)
    i+=1

plt.tight_layout()
```




They all seem to be leaking the target, so we should drop them just to be safe.

In [17]:

```
df.drop(physician_vars, axis=1, inplace=True)
```

**Form_of_Payment_or_Transfer_of_Value**

In [18]:

```
df.groupby(['research_payment', 'Form_of_Payment_or_Transfer_of_Value']).size().reset_index()
```

Out[18]:

| | research_payment | Form_of_Payment_or_Transfer_of_Value | 0 |
|---|---|---|---|
| **0** | 0 | Cash or cash equivalent | 3165 |
| **1** | 0 | Dividend, profit or other return on investment | 1 |
| **2** | 0 | In-kind items and services | 16834 |
| **3** | 1 | Cash or cash equivalent | 15834 |
| **4** | 1 | In-kind items and services | 4166 |

Again, the proportions are very different for each payment type and it seems to be very heavily correlated with the target but due to the nature of the variable we conclude that it's less likely that it leaks the target - it's more probably a very good predictor for the target.

#### Irrelevant Variables

After a quick scan of the remaining variables, we can remove those that don't or at least shouldn't have anything to do with the target.

In [19]:

```
cols_drop = ['Recipient_Primary_Business_Street_Address_Line2']
df.drop(cols_drop, axis=1, inplace=True)
obs_vars.append(cols_drop)
```

#### Remaining Variables

For the rest of the variables, since they have a lot of categories, we can't really plot them. Instead, we'll plot the missingness of the data depending on the target in the hopes to observe some form of data leakage (eg if a variable has a lot of missing values for target = 1 but not for target = 0).

In [20]:

```
obs_vars.append('research_payment')
unobs_vars = [x for x in df.columns.values if x not in obs_vars]
```

In [21]:

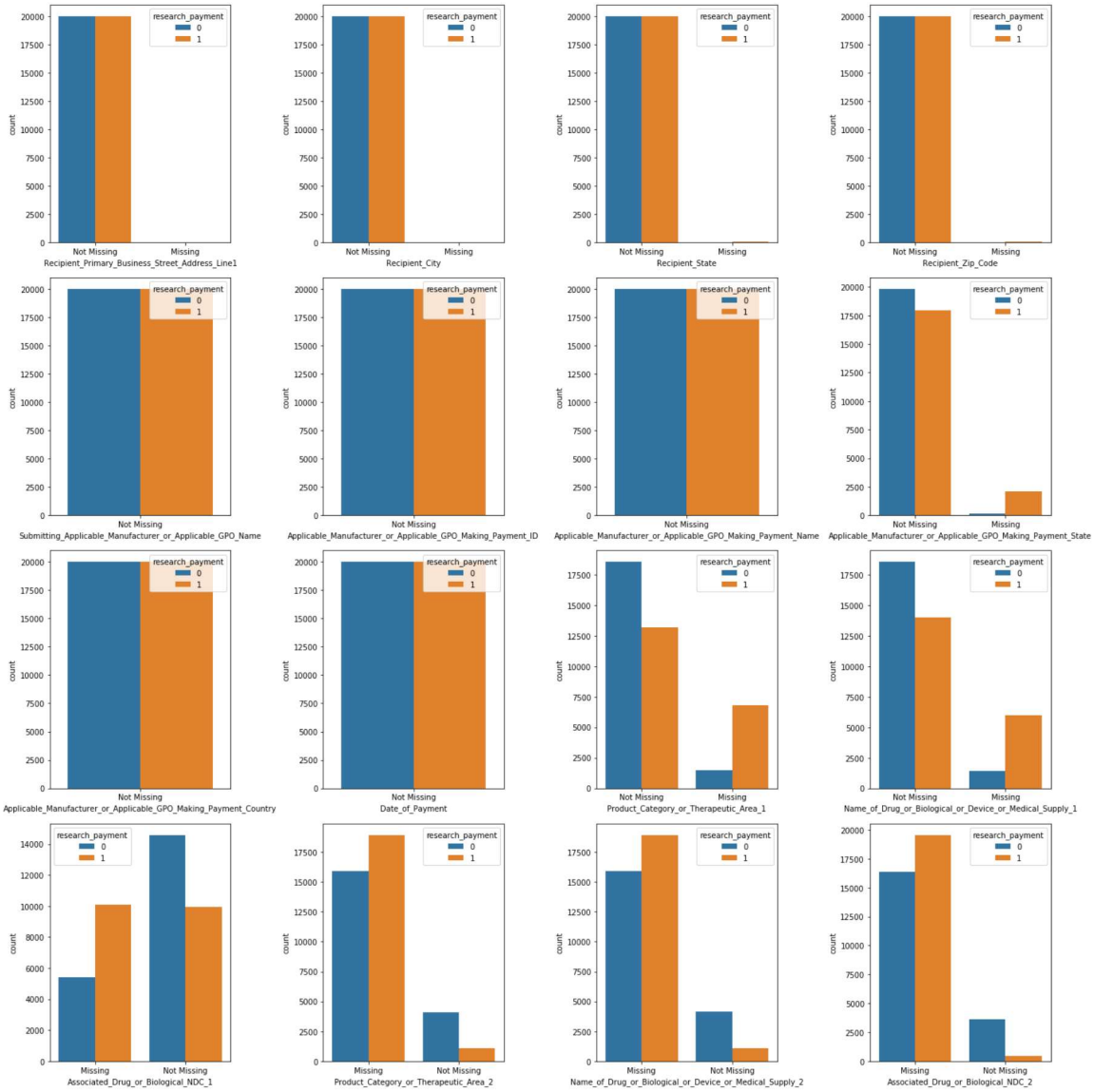
```
len(unobs_vars)
```

Out[21]:

16

In [22]:

```
fig, ax= plt.subplots(4, 4, figsize=(20, 20))
i=0
j=0

for v in unobs_vars:
    sns.countplot(np.where(df[v].notnull(), 'Not Missing', 'Missing'), hue=df['research
_payment'], ax=ax[j, i])
    ax[j, i].set_xlabel(v)
    i+=1
    if i == 4: j+=1;i=0

plt.tight_layout()
```




Nothing here is too shocking or surprising, so we drop none of those variables.

### Renaming our Variables

In [23]:

```
df = df_rename(df)
```

# Task 2: Preprocessing and baseline model

For the simple minimum viable model, we've decided to do logistic regression with a very limited number of features. The filter we've applied to features is the following (those are very "harsh" filters but again, this is a baseline model).

- Drop all features with more than 10% missing data (considered "sparse")
- Drop all categorical features with too many categories since we're going to one-hot encode them. We've chosen 200 (1% of the number of data points) as a threshold
- Drop the date feature as it's not in the appropriate format yet.

In [24]:

```
cols_drop2 = list(df.columns[df.isnull().sum() > 0.1*df.shape[0]])
cats = df.columns[df.dtypes == object]
cols_drop2.extend(cats[df[cats].nunique() > 0.01*df.shape[0]])
cols_drop2.append('payment_date')
df2 = df.drop(set(cols_drop2), axis=1)
df2.columns
```

Out[24]:

```
Index(['Change_Type', 'state', 'paying_GPO_state', 'paying_GPO_country',
       'payment_amount', 'payment_form', 'Related_Product_Indicator',
       'product1_covered', 'research_payment'],
      dtype='object')
```

As we can see, we're left with only a few features, only one of which is continuous. We can now set up a baseline model, with basic preprocessing: we're only going to one-hot encode the categorical variables.

We assign the "missing" category to missing values in categorical variables (such treatment normally shouldn't be done on X directly, but here it doesn't imply data leakage so it's OK). No treatment is needed for the continuous variable since it has no missing value. We then split training and test set.

We also need to change y to -1 and 1.

In [25]:

```
X = df2.drop('research_payment', axis=1)
y = df2['research_payment']

X_cats = X.columns[X.dtypes == object].tolist()
dict_missing = {i: 'Missing' for i in X_cats}
X.fillna(dict_missing, inplace=True)

X_train, X_test, y_train, y_test = train_test_split(X, y)

X_cats = X.dtypes == object
```

In [26]:

```
ohe = OneHotEncoder(handle_unknown = 'ignore')
logr = LogisticRegression(C=10000000)
prep = ColumnTransformer([("one-hot encoding", ohe, X_cats), ('pt', 'passthrough', ~X_c
ats)])
pipe = Pipeline([('preprocessing', prep), ('logreg', logr)])
np.mean(cross_val_score(pipe, X_train, y_train, cv=10, scoring='roc_auc'))
```

Out[26]:

```
0.9355582800096316
```

In [27]:

```
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)
```

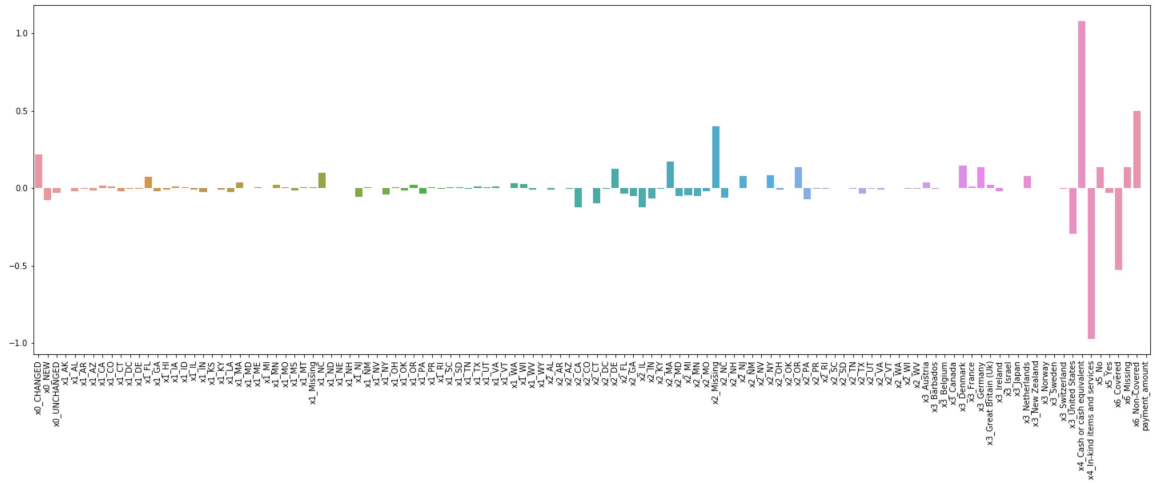
Out[27]:

```
0.8434
```

We get a surprisingly high cross-validation score but a much lower test score, suggesting that our model overfits. Here are the feature importances:

In [28]:

```
cols = pipe.named_steps['preprocessing'].named_transformers_['one-hot encoding'].get_fe
ature_names().tolist() + X_cats[X_cats == False].index.tolist()
fig, ax = plt.subplots(1, 1, figsize=(25, 8))
plt.xticks(rotation='vertical')
sns.barplot(x=cols, y=pipe.named_steps['logreg'].coef_[0])
plt.show()
```




We notice that the two highest features in absolute values are actually correlated, since they correspond to the two one-hot encoded categories of the feature "Form_of_payment". This is due to the fact that because this is supposed to be a baseline model, we did as little regularization as possible (large C), but this will get corrected later.

# Task 3: Feature Engineering

Building on a lot of the feature analysis done above, we can engineer our variables with a finer detail to enhance our model performance.

In [29]:

```
for v in ['paying_GPO_ID', 'payment_amount', 'research_payment']:
    df[v] = pd.to_numeric(df[v])
df['payment_date'] = pd.to_datetime(df.payment_date)
```

In [30]:

```
df.head(2)
```

Out[30]:

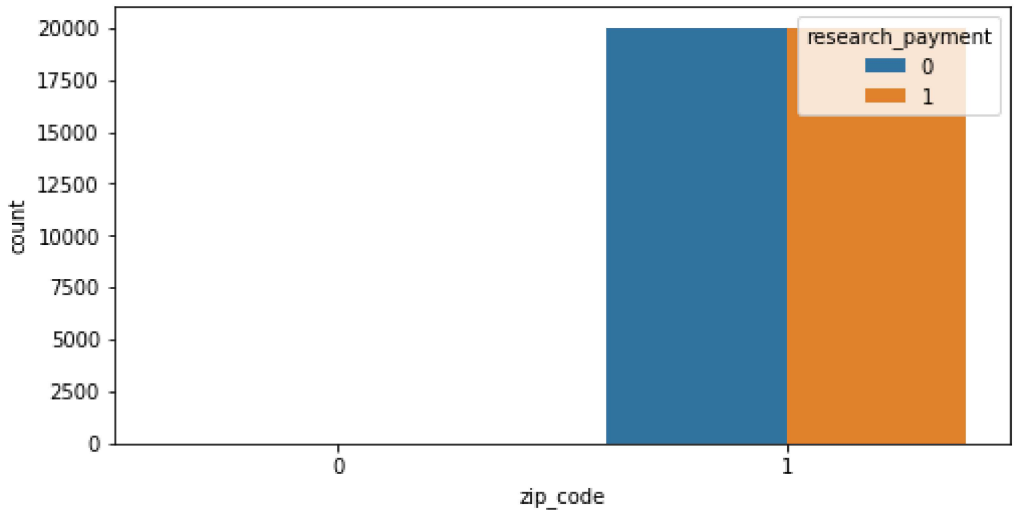
| | Change_Type | address | city | state | zip_code | submitting_GPO_name | payi |
|---|---|---|---|---|---|---|---|
| 0 | UNCHANGED | 4921 PARKVIEW PL | SAINT LOUIS | MO | 63110-1032 | Mevion_Medical_Systems_Inc | 10( |
| 1 | UNCHANGED | 1482 E WILLIAMS FIELD RD | GILBERT | AZ | 85295 | Bovie Medical Corporation | 10( |

Although the values of a lot of our variables provide little insight, the fact that they're non empty may be more useful.

**Zip-Code**

In [31]:

```
fig, ax = plt.subplots(1, 1, figsize=(8, 4))

sns.countplot(np.where(df.zip_code.notnull(), 1, 0), hue=df['research_payment'], ax=ax)
ax.set_xlabel('zip_code')
plt.show()
```



Zip-code has a unique value for almost all observations so it's safe to drop it.

In [32]:

```
df.drop(['zip_code'], axis=1, inplace=True)
```

For the remaining text variables, it might be a good idea to set all their values to lower case before proceeding in case there are any duplicates that have been skipped.

In [33]:

```
for c in df.columns.values[:-1]:
    if df[c].dtype == object:
        df[c] = df[c].str.lower()
```

### Address

Almost every observation has a different address, so on first glance this might not be a very useful variable.

In [34]:

```
df.groupby(['research_payment', 'address']).size().reset_index().iloc[np.r_[0:2, -2:0]]
```

Out[34]:

| | research_payment | address | 0 |
|---|---|---|---|
| **0** | 0 | 1 andrew ct | 1 |
| **1** | 0 | 1 associate dr | 1 |
| **23245** | 1 | yale university school of medicinetreasury ope... | 1 |
| **23246** | 1 | york st ynhh smilow cancer center | 1 |

It might be worth extracting the notable parts of each address line by looking for some keywords, e.g university, laboratory, medical, hospital etc.

We'll map all address to either a hospital/lab/uni or other.

In [35]:

```
# university
df['address'] = np.where(df.address.str.contains('uni|depart|medic|lab|resear|foundat',
regex=True), 'university', df.address)
# hospital
df['address'] = np.where(df.address.str.contains('hosp|clinic', regex=True), 'hospital'
, df.address)
# po box
df['address'] = np.where(df.address.str.contains('box', regex=True), 'pobox', df.addres
s)
# other
df['address'] = np.where(df.address.str.contains('hosp|lab|uni|box', regex=True), df.ad
dress, 'other')
```

In [36]:

```
df.groupby(['research_payment', 'address']).size().reset_index()
```

Out[36]:

| | research_payment | address | 0 |
|---|---|---|---|
| **0** | 0 | hospital | 201 |
| **1** | 0 | other | 18918 |
| **2** | 0 | pobox | 222 |
| **3** | 0 | university | 659 |
| **4** | 1 | hospital | 160 |
| **5** | 1 | other | 18200 |
| **6** | 1 | pobox | 608 |
| **7** | 1 | university | 1032 |

### City

In [37]:

```
print('The city variable has {} unique values.'.format(len(df.city.unique())))
```

```
The city variable has 3315 unique values.
```

In [38]:

```
df.groupby(['research_payment', 'city']).size().reset_index()[df.groupby(['research_pay
ment', 'city']).size().reset_index()[0] >= (df.shape[0] / 200)]
```

Out[38]:

| | research_payment | city | 0 |
|---|---|---|---|
| **1239** | 0 | houston | 296 |
| **1877** | 0 | new york | 311 |
| **3115** | 1 | atlanta | 258 |
| **3192** | 1 | boston | 420 |
| **3282** | 1 | chicago | 303 |
| **3348** | 1 | dallas | 496 |
| **3601** | 1 | houston | 418 |
| **3750** | 1 | los angeles | 345 |
| **3823** | 1 | miami | 241 |
| **3893** | 1 | nashville | 227 |
| **3911** | 1 | new york | 407 |
| **4026** | 1 | philadelphia | 242 |
| **4152** | 1 | san antonio | 233 |

It might be beneficial for us to keep the most popular cities and group all the others in a common category as the number of different categories certainly doesn't help our model.

In [39]:

```
popular_cities = df.groupby(['research_payment', 'city']).size().reset_index()[df.group
by(['research_payment', 'city']).size().reset_index()[0] >= (df.shape[0] / 200)]['city'
].unique()
# remap the city variable
df['city'] = np.where(np.isin(df.city, popular_cities), df.city, 'other')
```

In [40]:

```
print('The variable city now has {} unique values.'.format(len(df.city.unique())))
```

```
The variable city now has 12 unique values.
```

**State**

In [41]:

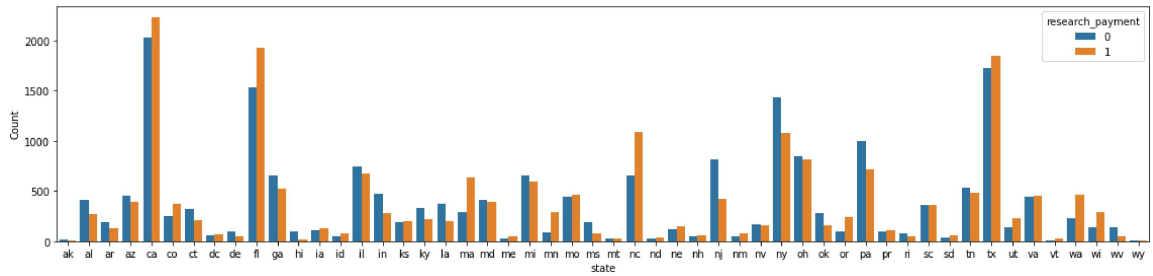
```
print('The variable State has {} unique values.'.format(len(df.state.unique())))
```

```
The variable State has 53 unique values.
```

Although there are not as many categories as in the previous variable, this might also be problematic. We can try to visualize the proportions in case it helps.

In [42]:

```
fig, ax = plt.subplots(1, 1, figsize=(16, 4))
sns.barplot(x='state', y=0, hue='research_payment', data=df.groupby(['research_payment'
, 'state']).size().reset_index(), ax = ax)
ax.set_ylabel('Count')
plt.tight_layout()
plt.show()
```



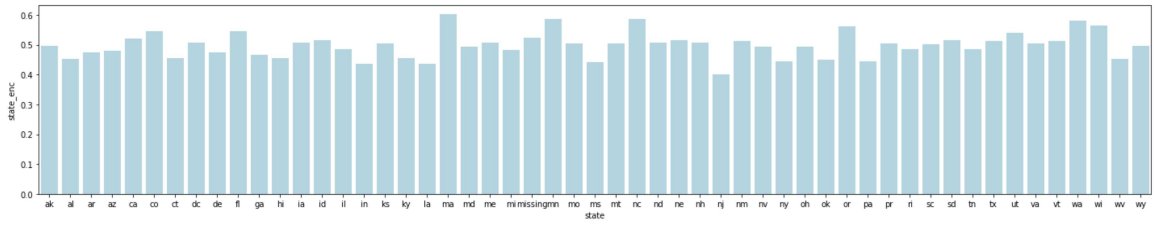

We could try Mean Encoding that may deal with that appropriately.

In [43]:

```
df['state'].fillna('missing', inplace=True)
df['state_enc'] = TargetEncoder().fit_transform(pd.DataFrame(df['state']), df['research
_payment'].values)
```

In [44]:

```
fig, ax = plt.subplots(1, 1, figsize=(20,4))
sns.barplot(x='state', y='state_enc', data=df.groupby('state')['state_enc'].first().res
et_index(), ax=ax, color='lightblue')
df.drop('state_enc', axis=1, inplace=True)
plt.tight_layout()
plt.show()
```

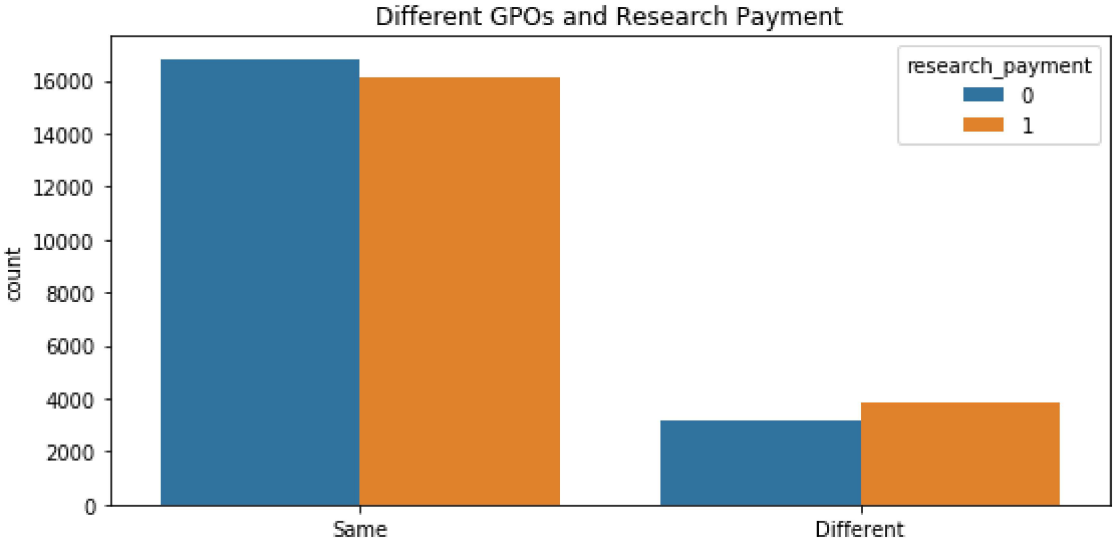


**GPO Names**

submitting_GPO_name and paying_GPO_name are very often the same; are the observations where the two are different indicative of anything?

In [45]:

```
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
sns.countplot(np.where(df.submitting_GPO_name != df.paying_GPO_name, 'Different', 'Sam
e'), hue=df['research_payment'])
ax.set_title('Different GPOs and Research Payment')
plt.xticks([0,1], labels=['Same', 'Different'])
plt.tight_layout()
plt.show()
```




It seems that the majority of payments have the same submitting and paying GPO; it also looks that the proportions are somewhat different for the two. The only information we'll retain from this is whether the submitting and payment GPOs are different.

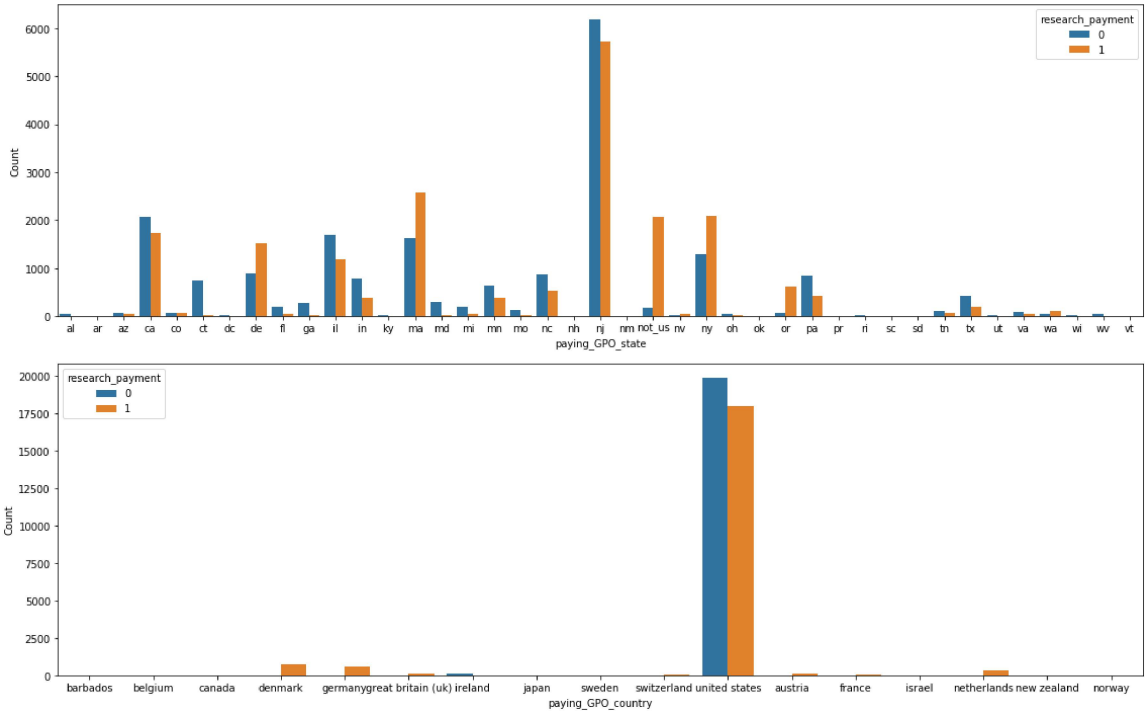
In [46]:

```
df['different_GPO'] = np.where(df.submitting_GPO_name != df.paying_GPO_name, 'Differen
t', 'Same')

# drop the original variables
df.drop(['submitting_GPO_name', 'paying_GPO_name', 'paying_GPO_ID'], axis=1, inplace=Tr
ue)
```

### Paying GPO State and Country

In [47]:

```
df.paying_GPO_state.fillna('not_us', inplace=True)

fig, ax = plt.subplots(2, 1, figsize=(16, 10))
sns.barplot(x='paying_GPO_state', y=0, hue='research_payment', data=df.groupby(['resear
ch_payment', 'paying_GPO_state']).size().reset_index(), ax = ax[0])
ax[0].set_ylabel('Count')
sns.barplot(x='paying_GPO_country', y=0, hue='research_payment', data=df.groupby(['rese
arch_payment', 'paying_GPO_country']).size().reset_index(), ax = ax[1])
ax[1].set_ylabel('Count')
plt.tight_layout()
plt.show()
```




In [48]:

```
df.drop('paying_GPO_country', axis=1, inplace=True)
```

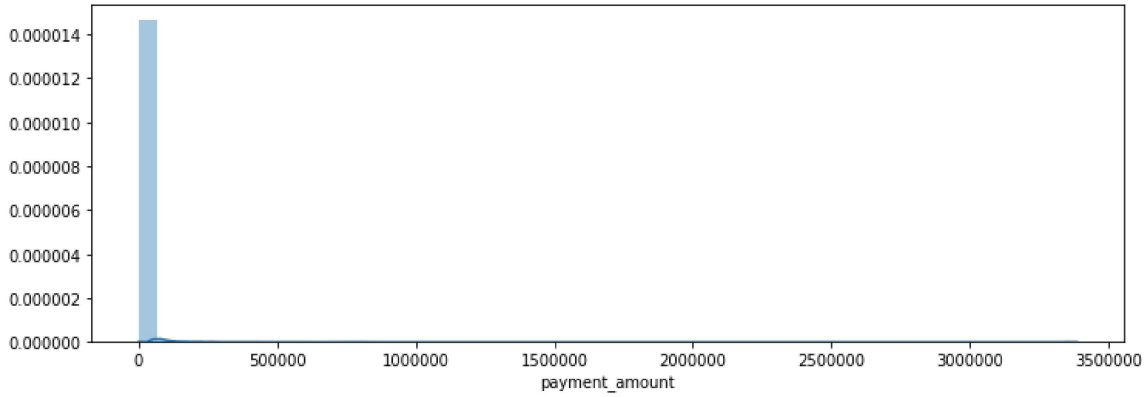
By incorporating the information provided in paying_GPO_country in paying_GPO_state, we can drop the country variable. We can now treat the paying GPO state variable the same way we did the original state variable, target encoding.

**Payment Amount**

Our only continuous variable, it's important to see whether it needs scaling or any other transformation.

In [49]:

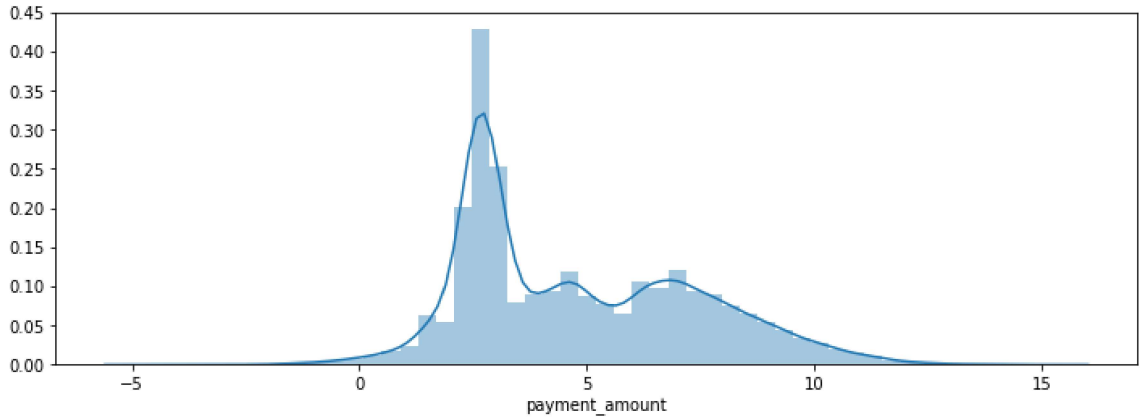
```
fig, ax = plt.subplots(1, 1, figsize=(12, 4))
sns.distplot(df.payment_amount, ax=ax)
plt.show()
```




A log transformation might fix the skewness.

In [50]:

```
fig, ax = plt.subplots(1, 1, figsize=(12, 4))
sns.distplot(np.log(df.payment_amount))
plt.show()
```




Although the transformation doesn't take care of the skewness completely, as there still seems to be a positive skewness in our data, we can still proceed.

In [51]:

```
df['log_payment_amount'] = np.log(df.payment_amount)
# drop the original
df.drop('payment_amount', axis=1, inplace=True)
```

**Payment Form**

In [52]:

```
df.groupby(['research_payment', 'payment_form']).size().reset_index()
```

Out[52]:

| | research_payment | payment_form | 0 |
|---|---|---|---|
| 0 | 0 | cash or cash equivalent | 3165 |
| 1 | 0 | dividend, profit or other return on investment | 1 |
| 2 | 0 | in-kind items and services | 16834 |
| 3 | 1 | cash or cash equivalent | 15834 |
| 4 | 1 | in-kind items and services | 4166 |

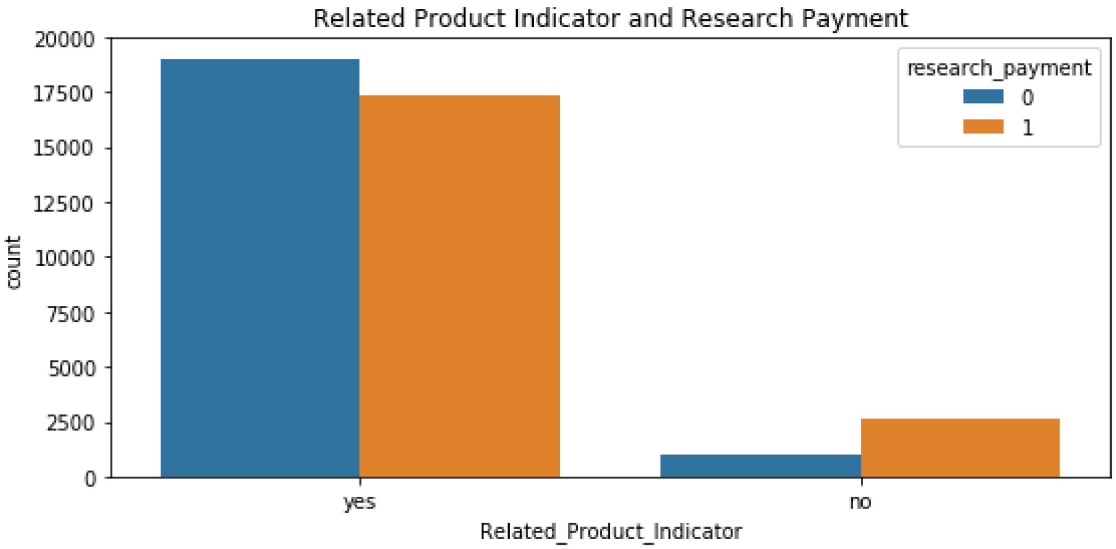
In [53]:

```
df['cash_payment'] = np.where(df.payment_form == 'cash or cash equivalent', 'cash', 'ot
her')
# drop the original
df.drop('payment_form', axis=1, inplace=True)
```

### Related Product Indicator

In [54]:

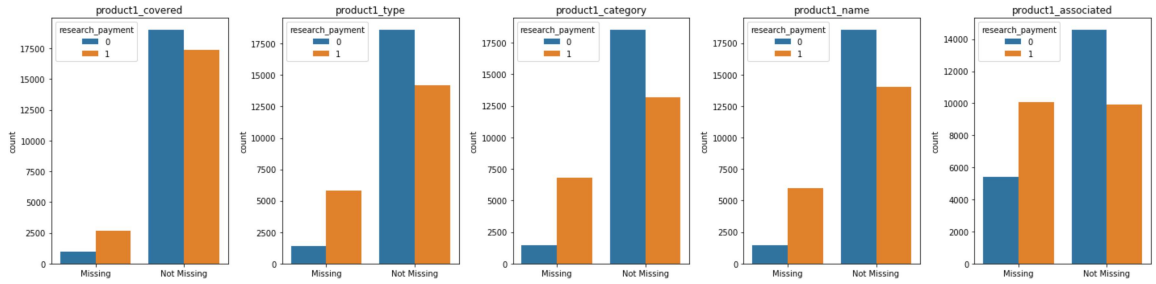
```
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
sns.countplot(df.Related_Product_Indicator, hue=df['research_payment'])
ax.set_title('Related Product Indicator and Research Payment')
plt.tight_layout()
plt.show()
```




### Product 1 Variables

In [55]:

```
p1_vars = ['product1_covered', 'product1_type', 'product1_category', 'product1_name',
'product1_associated']
fig, ax = plt.subplots(1, 5, figsize=(20, 5))

for i in range(len(p1_vars)):
    sns.countplot(np.where(df[p1_vars[i]].isnull(), 0, 1), hue=df['research_payment'],
ax=ax[i])
    ax[i].set_title(p1_vars[i])
    ax[i].set_xticklabels(['Missing', 'Not Missing'])

plt.tight_layout()
plt.show()
```




In [56]:

```
df.product1_covered.fillna('missing', inplace=True)
df.groupby(['research_payment', 'product1_covered']).size().reset_index()
```

Out[56]:

| | research_payment | product1_covered | 0 |
|---|---|---|---|
| **0** | 0 | covered | 18491 |
| **1** | 0 | missing | 992 |
| **2** | 0 | non-covered | 517 |
| **3** | 1 | covered | 12671 |
| **4** | 1 | missing | 2636 |
| **5** | 1 | non-covered | 4693 |

In [57]:

```
df.product1_type.fillna('missing', inplace=True)
df.groupby(['research_payment', 'product1_type']).size().reset_index()
```

Out[57]:

| | research_payment | product1_type | 0 |
|---|---|---|---|
| **0** | 0 | biological | 2026 |
| **1** | 0 | device | 3767 |
| **2** | 0 | drug | 12799 |
| **3** | 0 | medical supply | 17 |
| **4** | 0 | missing | 1391 |
| **5** | 1 | biological | 2951 |
| **6** | 1 | device | 2242 |
| **7** | 1 | drug | 8965 |
| **8** | 1 | medical supply | 2 |
| **9** | 1 | missing | 5840 |

In [58]:

```
df.product1_type.fillna('missing', inplace=True)
df.groupby(['research_payment', 'product1_category']).size().reset_index().iloc[np.r_[0
:2, -2:0]]
```

Out[58]:

| | research_payment | product1_category | 0 |
|---|---|---|---|
| **0** | 0 | 1 | 12 |
| **1** | 0 | ablation | 4 |
| **892** | 1 | woundcare | 1 |
| **893** | 1 | x-ray | 2 |

This feature also requires further parsing.

In [59]:

```
df['product1_category'] = df['product1_category'].str.split(' ').str[0]
```

We keep the most popular categories and group the rest as 'other'.

In [60]:

```
p1_cats = df.groupby(['research_payment', 'product1_category']).size().reset_index()[df
.groupby(['research_payment', 'product1_category']).size().reset_index()[0] > 200]['pro
duct1_category'].unique()

# remap p1 category
df['product1_category'] = np.where(np.isin(df.product1_category, p1_cats), df.product1_
category, 'other')
```

In [61]:

```
df.product1_name.fillna('missing', inplace=True)

df.groupby(['research_payment', 'product1_name']).size().reset_index().iloc[np.r_[0:2,
-2:0]]
```

Out[61]:

| | research_payment | product1_name | 0 |
|---|---|---|---|
| **0** | 0 | (810) chenodal | 1 |
| **1** | 0 | (815) thiola | 4 |
| **2595** | 1 | zykadia | 5 |
| **2596** | 1 | zytiga | 10 |

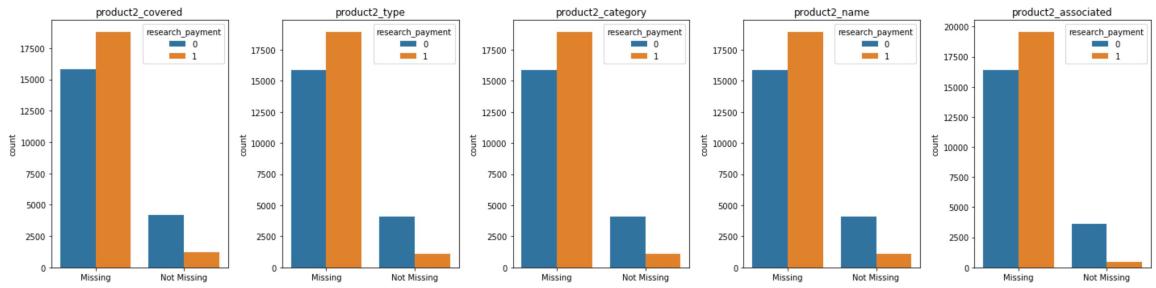
In [62]:

```
df.drop(['product1_name', 'product1_associated'], axis=1, inplace=True)
```

### Product 2 Variables

In [63]:

```
p2_vars = ['product2_covered', 'product2_type', 'product2_category', 'product2_name',
'product2_associated']
fig, ax = plt.subplots(1, 5, figsize=(20, 5))

for i in range(len(p2_vars)):
    sns.countplot(np.where(df[p2_vars[i]].isnull(), 0, 1), hue=df['research_payment'],
ax=ax[i])
    ax[i].set_title(p2_vars[i])
    ax[i].set_xticklabels(['Missing', 'Not Missing'])

plt.tight_layout()
plt.show()
```



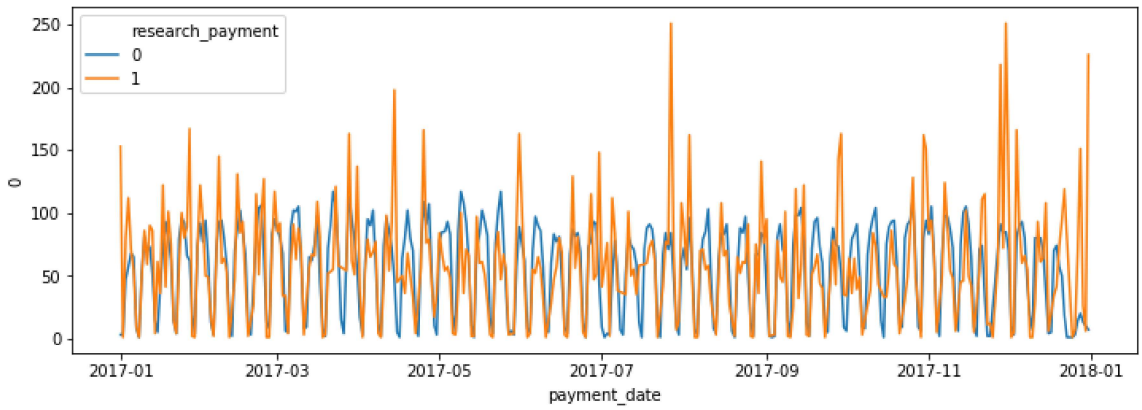
In [64]:

```
df['p2_missing'] = np.where(df.product2_covered.isnull(), 'missing', 'not_missing')

# drop the original
df.drop(p2_vars, axis=1, inplace=True)
```

**Payment Date**

In [65]:

```
fig, ax = plt.subplots(1,1, figsize=(12,4))
sns.lineplot(x='payment_date', y=0, hue='research_payment', data=df.groupby(['research_
payment', 'payment_date']).size().reset_index())
plt.show()
```




To get as much information as possible from the payment date variable, we can extract the month, year, day of the month and weekday.

In [66]:

```
# day of the week
df['payment_weekday'] = df.payment_date.dt.weekday

# month
df['payment_month'] = df.payment_date.dt.month

# day of year
df['payment_dayofyear'] = df.payment_date.dt.dayofyear

# drop the original
df.drop('payment_date', axis=1, inplace=True)
```

In [67]:

```
df.head(2)
```

Out[67]:

| | Change_Type | address | city | state | paying_GPO_state | Related_Product_Indicator | produc |
|---|---|---|---|---|---|---|---|
| **0** | unchanged | other | other | mo | ma | yes | |
| **1** | unchanged | other | other | az | fl | no | |

# Task 4: Any model

The models we've decided to use for this part are the following:

- Ridge Logistic regression, as an extension of the baseline model
- Kernelized SVM (RBF Kernel)
- Random Forests
- XGBoost

We will do some preprocessing for Logistic Regression, SVM and Random Forests using Column Transformers, which we'll include in a pipeline. Let's first get out training and test sets:

In [68]:

```
# get X and y
y = df.research_payment
X = df.drop('research_payment', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

## Preprocessing

We have three column transformers:

- One for your categorical variables that need to be one hot encoded.
- One for the categorical variables that need to be mean encoded (State and paying GPO state)
- And one for our continuous and ordinal variables (payment amount, month and day of year): we'll apply a standard scaler. This scaler will only be applied for models that are sensitive to scaling: LogReg and SVM.

Please note that we don't have to worry about imputing any missing values, because our previous treatment has left no missing value at all.

In [69]:

```
ohe = OneHotEncoder(categories="auto", handle_unknown = 'ignore')
rsc = RobustScaler()
menc = TargetEncoder(handle_unknown = 'ignore')
cat_mask = X_train.dtypes == object
cont_mask = ~cat_mask
ord_mask = X_train.columns.isin(['state', 'paying_GPO_state'])
cat_mask = np.logical_and(cat_mask, ~ord_mask)

prepsc = ColumnTransformer([('one-hot encoding', ohe, cat_mask), ('mean encoding', menc
, ord_mask), ('scaling', rsc, cont_mask)])
prepnsc = ColumnTransformer([('one-hot encoding', ohe, cat_mask), ('mean encoding', men
c, ord_mask), ('pt', 'passthrough', cont_mask)])
```

## Ridge Logistic Regression

We start with Logistic Regression. We apply a penalty this time (L2, because we have no reason to think that the solution is sparse). The grid-search is done on C.

In [70]:

```
logr = LogisticRegression()
pipe_logr = Pipeline([('preprocessing', prepsc), ('logr', logr)])
param_grid_logr = {'logr__C': np.logspace(-3, 2, 6)}

grid_logr = GridSearchCV(pipe_logr, param_grid_logr, cv=10, n_jobs=-1).fit(X_train, y_t
rain)
```

Let's display a basic evaluation of the model: the best score found in the grid search (the score being precision), the score curve depending on C, and the best model's score on the test set.

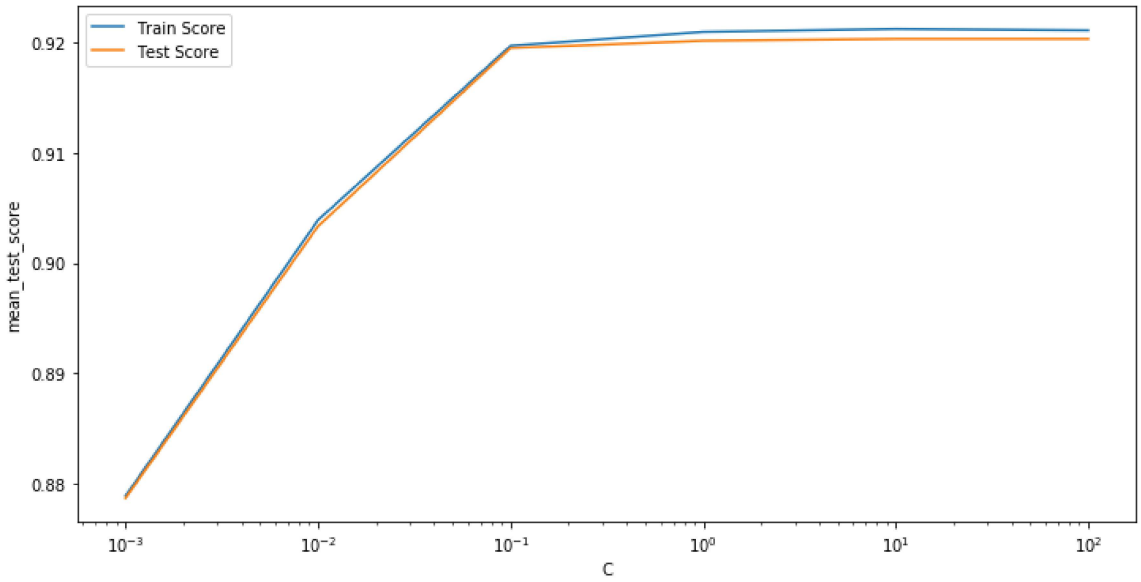
In [71]:

```
s_logr = grid_logr.best_score_
print("Best score for Logistic Regression: " + str(np.round(s_logr,2)))
```

```
Best score for Logistic Regression: 0.92
```

In [72]:

```
res = pd.DataFrame(grid_logr.cv_results_)

fig, ax = plt.subplots(1, 1, figsize=(12,6))

sns.lineplot(x='param_logr__C', y='mean_train_score', data=res, label='Train Score')
sns.lineplot(x='param_logr__C', y='mean_test_score', data=res, label='Test Score')

ax.set_xlabel('C')
plt.xscale("log")
plt.legend()
plt.show()
```




In [73]:

```
ts_logr = grid_logr.best_estimator_.score(X_test, y_test)
print("Test score for Logistic Regression: " + str(ts_logr))
```

```
Test score for Logistic Regression: 0.922
```

## Kernelized SVM

We now do Kernelized SVM. We decided to do a Grid Search on both important parameters, C (penalty) and gamma (rbf kernel coefficient).

In [74]:

```
svc = SVC()
pipe_svc = Pipeline([('preprocessing', prepsc), ('svc', svc)])
param_grid_svc = {'svc__C': np.logspace(-3, 2, 6),
                  'svc__gamma': np.logspace(-3, 2, 6)}

grid_svc = GridSearchCV(pipe_svc, param_grid_svc, cv=5, n_jobs=-1)
_ = grid_svc.fit(X_train, y_train)
```

Let's display a basic evaluation of the model: the best score found in the grid search (the score being precision) and the score matrix depending on C and gamma, as well as the test set score of the best estimator in the gridsearch.

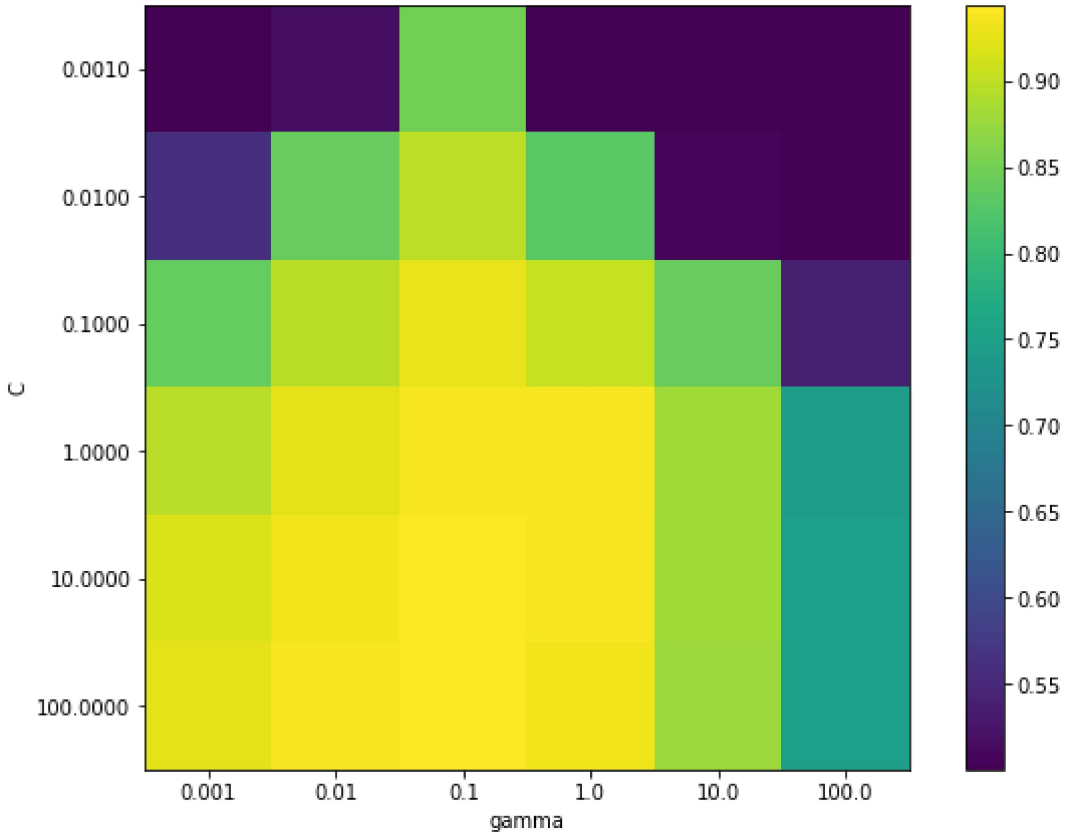
In [75]:

```
print("Best score for Kernelized SVM: {}".format(np.round(grid_svc.best_score_, 2)))
```

```
Best score for Kernelized SVM: 0.94
```

In [76]:

```
res = pd.pivot_table(pd.DataFrame(grid_svc.cv_results_),
                     values='mean_test_score', index='param_svc__C',
                     columns='param_svc__gamma')
plt.figure(figsize = (10,7))
plt.imshow(res)
plt.colorbar()
Cs = param_grid_svc['svc__C']
gammas = np.array(param_grid_svc['svc__gamma'])
plt.xlabel("gamma")
plt.ylabel("C")
plt.yticks(range(len(Cs)), ["{:.4f}".format(a) for a in Cs])
plt.xticks(range(len(gammas)), gammas);
```




In [77]:

```
ts_svc = grid_svc.best_estimator_.score(X_test, y_test)
print("Test score for Kernelized SVM: " + str(ts_svc))
```

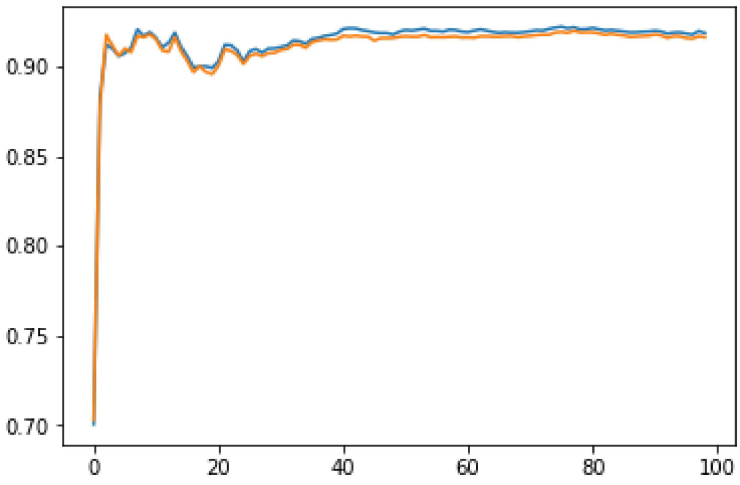
```
Test score for Kernelized SVM: 0.9403
```

## Random Forests

For random forests, we first try a warm-start with default parameters to determine how many classifiers we should use for the model. We set max-depth now to limit computation time and the size of the model: we arbitrarily decide on a max-depth of 5.

In [78]:

```
train_scores = []
test_scores = []
rf = RandomForestClassifier(warm_start=True, max_depth=5, random_state=0)
estimator_range = range(1, 100)
X_trainp = prepnsc.fit_transform(X_train, y_train)
X_testp = prepnsc.fit_transform(X_test, y_test)

for n_estimators in estimator_range:
    rf.n_estimators = n_estimators
    rf.fit(X_trainp, y_train)
    train_scores.append(rf.score(X_trainp, y_train))
    test_scores.append(rf.score(X_testp, y_test))

plt.plot(train_scores)
plt.plot(test_scores)
plt.show()
```




As we can see, the benefits of adding classifiers strongly diminish once we're past 25, so that's the value we'll keep.

We then do the Grid-Search on max-features: we try from 5 to 15. We have around 60 features after one-hot encoding and we know max-features should be around the square root of that, which is around 8.

In [79]:

```
# classifier and pipeline
rf = RandomForestClassifier(n_estimators = 25)
pipe_rf = Pipeline([('preprocessing', prepnsc), ('rf', rf)])

# param search
param_grid_rf = {'rf__max_depth': np.arange(5, 15, 1)}

# grid search
grid_rf = GridSearchCV(pipe_rf, param_grid_rf, cv=10, n_jobs=-1).fit(X_train, y_train)
```

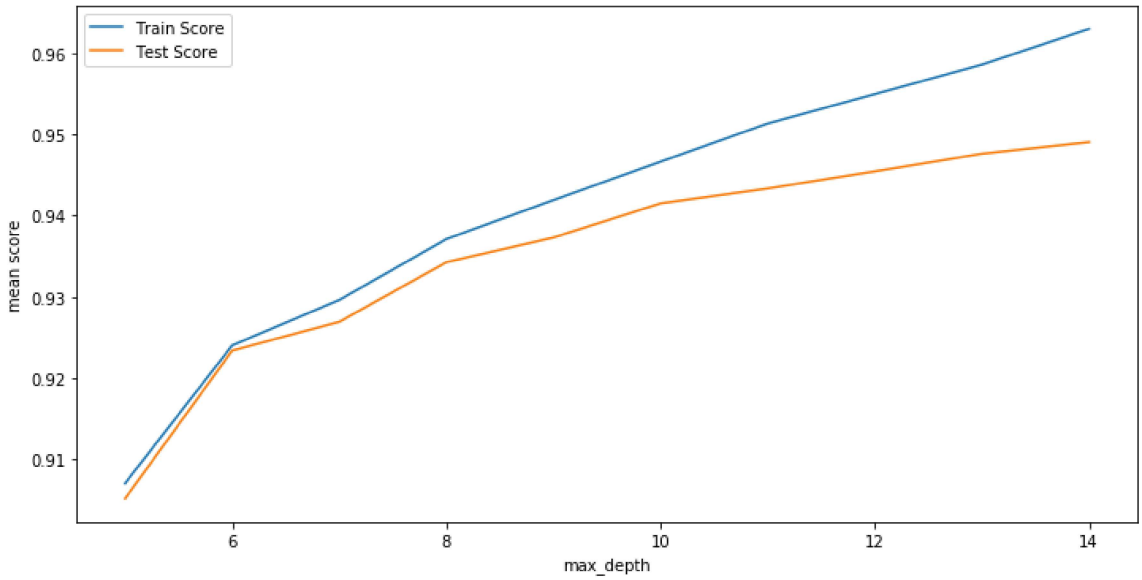
In [80]:

```
print("Best score for Random Forests: {}".format(np.round(grid_rf.best_score_, 2)))
```

```
Best score for Random Forests: 0.95
```

In [81]:

```
res = pd.DataFrame(grid_rf.cv_results_)

fig, ax = plt.subplots(1, 1, figsize=(12,6))

sns.lineplot(x='param_rf__max_depth', y='mean_train_score', data=res, label='Train Scor
e')
sns.lineplot(x='param_rf__max_depth', y='mean_test_score', data=res, label='Test Score'
)

ax.set_xlabel('max_depth')
ax.set_ylabel('mean score')
plt.legend()
plt.show()
```




The training score increases with max_depth which we expected, but we can see that the test score increases too.

In [82]:

```
ts_rf = grid_rf.best_estimator_.score(X_test, y_test)
print("Test score for Random Forests: " + str(ts_rf))
```

```
Test score for Random Forests: 0.9467
```

## XGBoost

Finally, we try an XGBoost model. The only change we'd have done with preprocessing would have been to remove the treatment of missing values, but we don't have missing values in the first place because they were all in categorical features and we imputed them as "missing". So we use the same preprocessing again.

We set the number of classifiers to 25.

We then do the Grid Search on the learning rate and on max-depth.

In [83]:

```
# classifier and pipeline
xgb = XGBClassifier(n_estimators = 25)
pipe_xgb = Pipeline([('preprocessing', prepnsc), ('xgb', xgb)])

# param search
param_grid_xgb = {'xgb__learning_rate': np.logspace(-2, 0, 5),
                  'xgb__max_depth': np.arange(5, 15, 1)}

# grid search
grid_xgb = GridSearchCV(pipe_xgb, param_grid_xgb, cv=10, n_jobs=2).fit(X_train, y_train
)
```

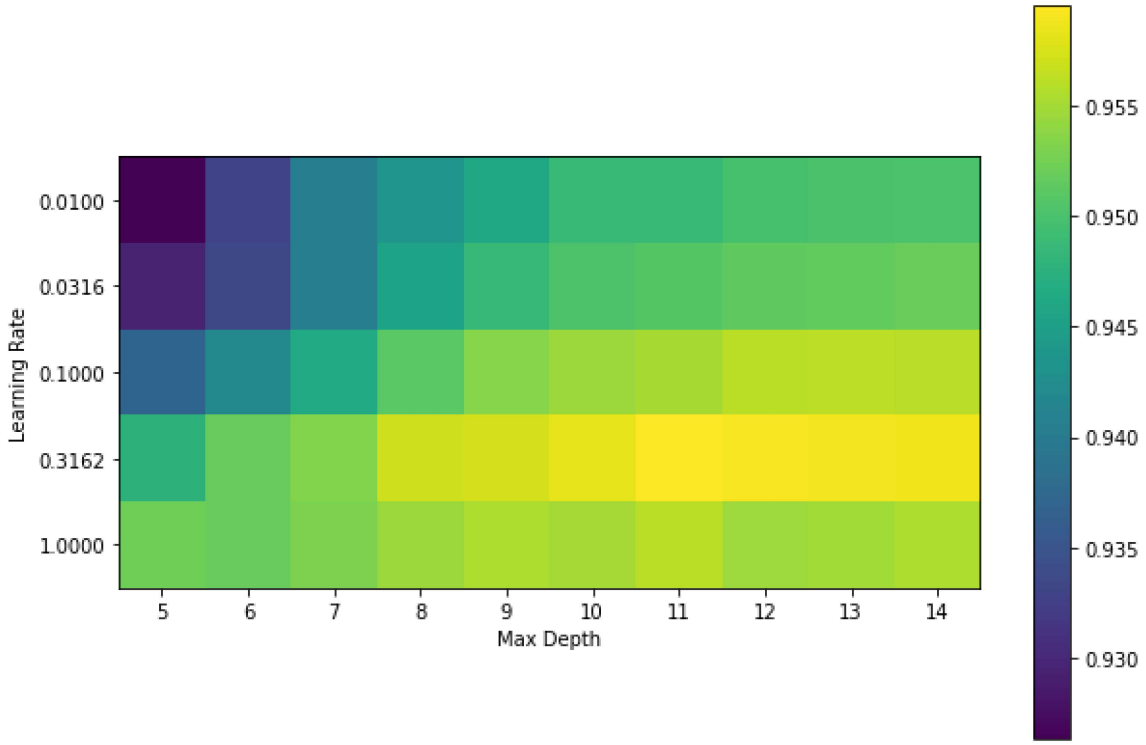
In [84]:

```
print("Best score for XGBoost: {}".format(np.round(grid_xgb.best_score_, 2)))
```

```
Best score for XGBoost: 0.96
```

In [85]:

```
res = pd.pivot_table(pd.DataFrame(grid_xgb.cv_results_),
                     values='mean_test_score', index='param_xgb__learning_rate',
                     columns='param_xgb__max_depth')
plt.figure(figsize = (10,7))
plt.imshow(res)
plt.colorbar()
lrs = param_grid_xgb['xgb__learning_rate']
md = np.array(param_grid_xgb['xgb__max_depth'])
plt.xlabel("Max Depth")
plt.ylabel("Learning Rate")
plt.yticks(range(len(lrs)), ["{:.4f}".format(a) for a in lrs])
plt.xticks(range(len(md)), md);
```




In [86]:

```
ts_xgb = grid_xgb.best_estimator_.score(X_test, y_test)
print("Test score for XGBoost: " + str(ts_xgb))
```

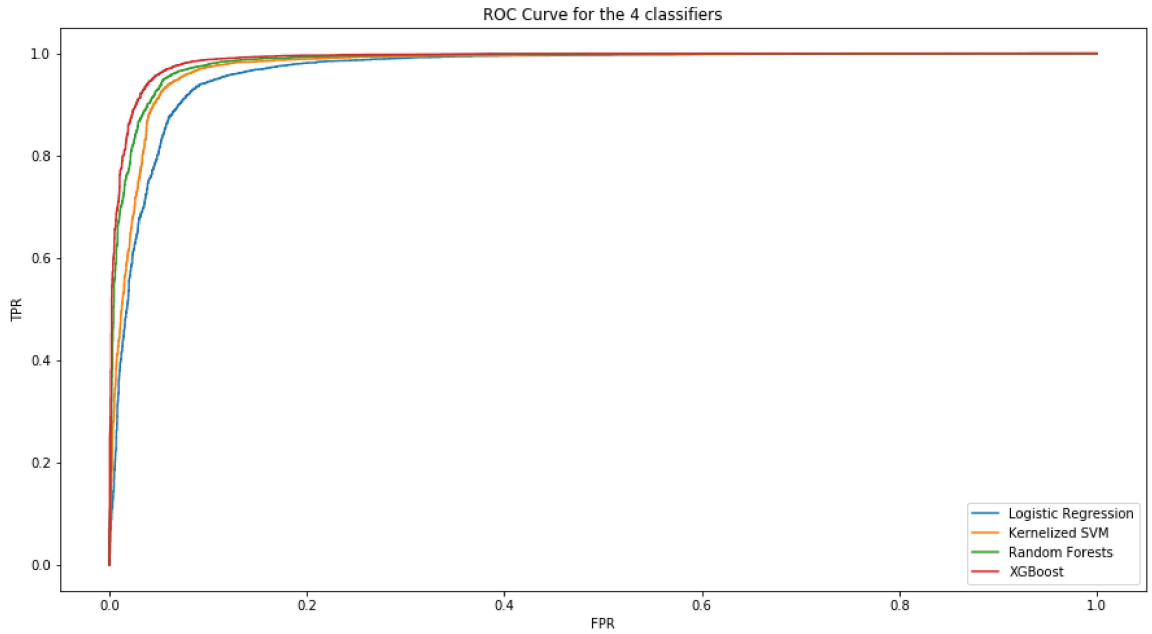
```
Test score for XGBoost: 0.9547
```

## Evaluating our models

As we can see from "simple" metrics (accuracy on test set), the best model from the four we tried is XGBoost. However, we want more detail in our model evaluation. Here, the assumption we make on the cost of making mistakes is that we don't associate different costs to false negatives and false positives - meaning that precision is actually a good metric. Still, we want to plot the ROC curve as well as the Precision/Recall curve for our four models, which are good general-purpose ways to visualize the performances of our models better.

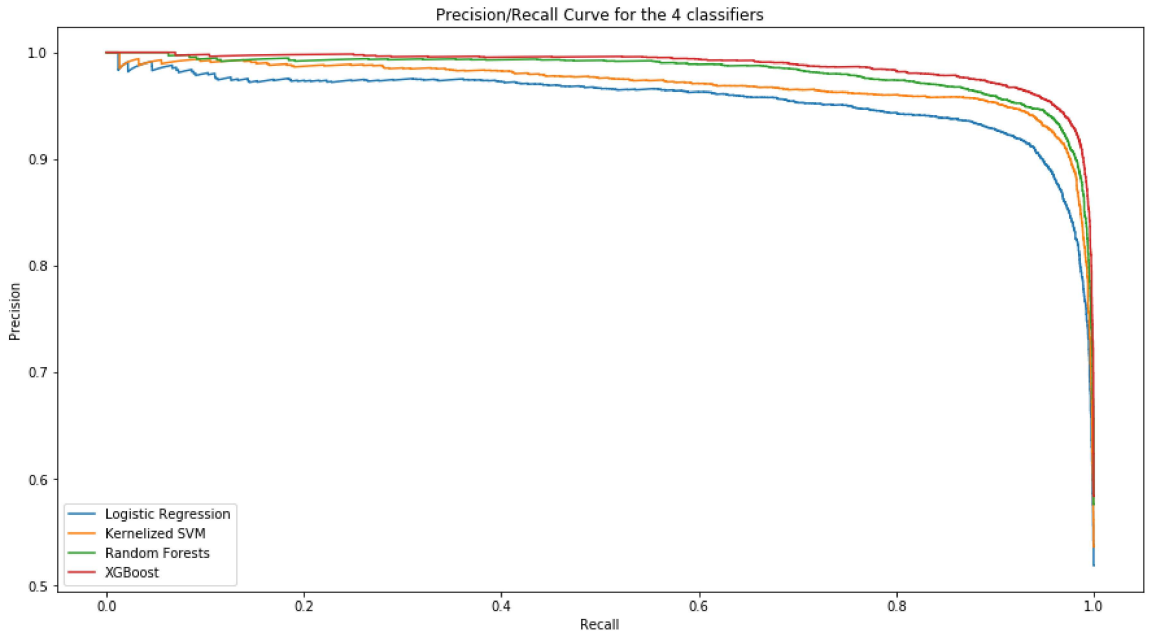
In [87]:

```
plt.figure(figsize = (15,8))
fpr_logr, tpr_logr, thresholds_logr = roc_curve(y_test, grid_logr.best_estimator_.decis
ion_function(X_test))
fpr_svc, tpr_svc, thresholds_svc = roc_curve(y_test, grid_svc.best_estimator_.decision_
function(X_test))
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, grid_rf.best_estimator_.predict_proba
(X_test)[:,1])
fpr_xgb, tpr_xgb, thresholds_xgb = roc_curve(y_test, grid_xgb.best_estimator_.predict_p
roba(X_test)[:,1])

plt.plot(fpr_logr, tpr_logr, label = 'Logistic Regression')
plt.plot(fpr_svc, tpr_svc, label = 'Kernelized SVM')
plt.plot(fpr_rf, tpr_rf, label = 'Random Forests')
plt.plot(fpr_xgb, tpr_xgb, label = 'XGBoost')

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.title("ROC Curve for the 4 classifiers")
plt.legend()
plt.show()
```



In [88]:

```
plt.figure(figsize = (15,8))
p_logr, r_logr, thresholds_logr = precision_recall_curve(y_test, grid_logr.best_estimat
or_.decision_function(X_test))
p_svc, r_svc, thresholds_svc = precision_recall_curve(y_test, grid_svc.best_estimator_.
decision_function(X_test))
p_rf, r_rf, thresholds_rf = precision_recall_curve(y_test, grid_rf.best_estimator_.pred
ict_proba(X_test)[:,1])
p_xgb, r_xgb, thresholds_xgb = precision_recall_curve(y_test, grid_xgb.best_estimator_.
predict_proba(X_test)[:,1])

plt.plot(r_logr, p_logr, label = 'Logistic Regression')
plt.plot(r_svc, p_svc, label = 'Kernelized SVM')
plt.plot(r_rf, p_rf, label = 'Random Forests')
plt.plot(r_xgb, p_xgb, label = 'XGBoost')

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision/Recall Curve for the 4 classifiers")
plt.legend()
plt.show()
```




What was already apparent with the scores is confirmed thanks to these curves: the order of our classifiers seems to consistently be XGBoost > Random Forests > Kernelized SVM > Logistic Regression. We will then keep XGBoost as our best model.

# Task 5: Feature Selections

In [89]:

```
model = grid_xgb.best_estimator_
```

In [90]:

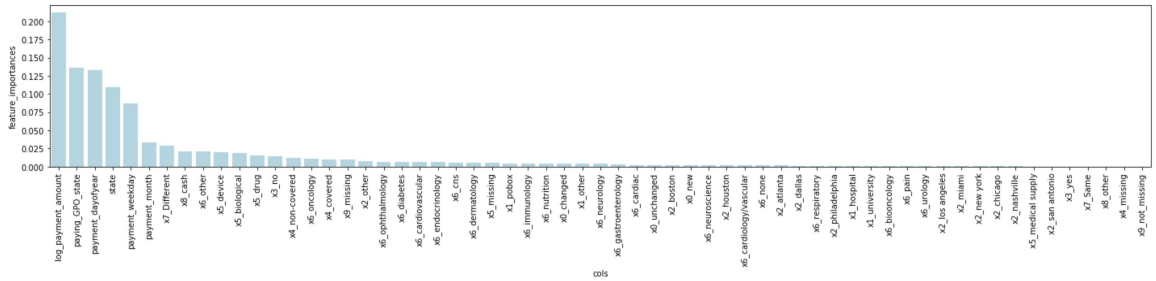
```
# get the encoded col_names
cols = model.named_steps['preprocessing'].named_transformers_['one-hot encoding'].get_f
eature_names().tolist()
# get the ordinal vars
cols.extend(X.columns[ord_mask])
# add the continuous vars
cols.extend(X.columns.values[cont_mask].tolist())
```

In [91]:

```
model_res = pd.DataFrame(data={'cols': cols,
                               'feature_importances': model.named_steps['xgb'].feature_
importances_})

model_res.sort_values(by='feature_importances', ascending=False, inplace=True)
```

In [92]:

```
fig, ax = plt.subplots(1, 1, figsize=(20, 5))
sns.barplot(x='cols', y='feature_importances', data=model_res, ax=ax, color='lightblue'
)
plt.xticks(rotation='vertical')
plt.tight_layout()
plt.show()
```




Judging by the results of the XGB model, it seems that the five most important variables are the following:.

- Payment Amount: the amount of the payment made.
- The state of the paying GPO: it seems that the state of the GPO that made the payment played an important role.
- The day of the year in which the payment was made also seemed to have a big influence (maybe the researchers were more concentrated around a certain semester ?)
- The state of the recipient, like the state of the paying GPO, was also important.
- Finally, the payment weekday also had a big influence - researchers might be more concentrated around weekdays and less around weekends, for instance

We can try dropping a few variables to see how that influences the performance of our model.

In [93]:

```
for t in np.logspace(-3, -1, 5):
    feat_selection = SelectFromModel(model.named_steps['xgb'], threshold=t)
    model_fs = Pipeline([('preprocessing', prepnsc),
                         ('fs', feat_selection),
                         ('xgb', XGBClassifier(n_estimators=25,
                                               learning_rate=grid_xgb.best_params_['xgb
__learning_rate'],
                                               max_depth=grid_xgb.best_params_['xgb__ma
x_depth']))]).fit(X_train, y_train)
    print('Test score of Feature Selection w/ Threshold {} is {}.'.format(np.round(t, 5
),
                                                                          np.round(mode
l_fs.score(X_test, y_test), 2)))
```

```
Test score of Feature Selection w/ Threshold 0.001 is 0.95.
Test score of Feature Selection w/ Threshold 0.00316 is 0.95.
Test score of Feature Selection w/ Threshold 0.01 is 0.95.
Test score of Feature Selection w/ Threshold 0.03162 is 0.92.
Test score of Feature Selection w/ Threshold 0.1 is 0.92.
```

Judging by the results of the Feature Selection, we could use a threshold of approximately 0.01 without sacrificing performance. This would mean that we eliminate the following variables:

In [94]:

```
model_res[model_res.feature_importances < 0.01]
```

Out[94]:

| | cols | feature_importances |
|---|---|---|
| **21** | x4_covered | 0.009730 |
| **53** | x9_missing | 0.009528 |
| **16** | x2_other | 0.007906 |
| **44** | x6_ophthalmology | 0.006892 |
| **35** | x6_diabetes | 0.006690 |
| **32** | x6_cardiovascular | 0.006284 |
| **36** | x6_endocrinology | 0.006081 |
| **33** | x6_cns | 0.005271 |
| **34** | x6_dermatology | 0.004865 |
| **28** | x5_missing | 0.004865 |
| **5** | x1_pobox | 0.004460 |
| **38** | x6_immunology | 0.004257 |
| **42** | x6_nutrition | 0.004054 |
| **0** | x0_changed | 0.004054 |
| **4** | x1_other | 0.004054 |
| **39** | x6_neurology | 0.003852 |
| **37** | x6_gastroenterology | 0.002838 |
| **30** | x6_cardiac | 0.002433 |
| **2** | x0_unchanged | 0.002433 |
| **8** | x2_boston | 0.002433 |
| **1** | x0_new | 0.002433 |
| **40** | x6_neuroscience | 0.002230 |
| **11** | x2_houston | 0.002027 |
| **31** | x6_cardiology/vascular | 0.001824 |
| **41** | x6_none | 0.001824 |
| **7** | x2_atlanta | 0.001622 |
| **10** | x2_dallas | 0.001419 |
| **47** | x6_respiratory | 0.001419 |
| **17** | x2_philadelphia | 0.001419 |
| **3** | x1_hospital | 0.001216 |
| **6** | x1_university | 0.001216 |
| **29** | x6_biooncology | 0.000811 |
| **46** | x6_pain | 0.000608 |
| **48** | x6_urology | 0.000608 |
| **12** | x2_los angeles | 0.000608 |
| **13** | x2_miami | 0.000608 |
| **15** | x2_new york | 0.000608 |

| | cols | feature_importances |
|---|---|---|
| **9** | x2_chicago | 0.000405 |
| **14** | x2_nashville | 0.000405 |
| **27** | x5_medical supply | 0.000203 |
| **18** | x2_san antonio | 0.000203 |
| **20** | x3_yes | 0.000000 |
| **50** | x7_Same | 0.000000 |
| **52** | x8_other | 0.000000 |
| **22** | x4_missing | 0.000000 |
| **54** | x9_not_missing | 0.000000 |

Removing these however does not increase the performance of our model, but does to an extent increase the interpretability due to the smaller feature space.

## Task 6: An explainable model

Here, we've chosen to build a decision tree as our simple, explainable model. We want our model to be relatively straightforward, so we set max_leaf_nodes to 8 (which gives us a max depth of 3 if we have a full binary tree). We don't do a grid-search since if we did one it would either be on max_leaf_nodes or max_depth: since we've already set max_leaf_nodes, we don't need to grid-search on max_depth since it's going to be limited anyway. Let's first build our model and check out the test score and ROC curve:

In [95]:

```
prepnsc.fit(X_train, y_train)
```

Out[95]:

```
ColumnTransformer(n_jobs=None, remainder='drop', sparse_threshold=0.3,
         transformer_weights=None,
         transformers=[('one-hot encoding', OneHotEncoder(categorical_feat
ures=None, categories='auto',
       dtype=<class 'numpy.float64'>, handle_unknown='ignore',
       n_values=None, sparse=True), Change_Type                   True
address                         True
city                            True
s...             True
payment_month                   True
payment_dayofyear               True
dtype: bool)])
```

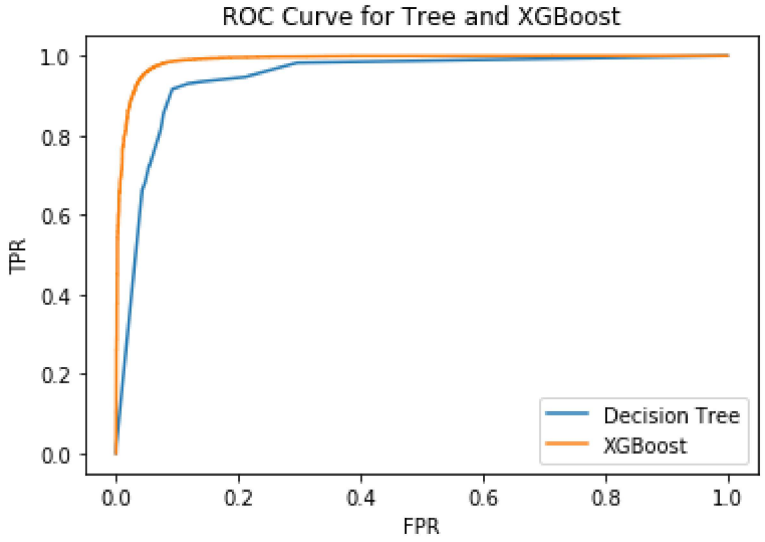
In [96]:

```
X_trainp = prepnsc.fit_transform(X_train, y_train)
X_testp = prepnsc.fit_transform(X_test, y_test)
tree = DecisionTreeClassifier(max_leaf_nodes=16, random_state=0).fit(X_trainp, y_train)
ts_tree = tree.score(X_testp, y_test)
print("Test score for Decision Tree: " + str(ts_tree))
```

```
Test score for Decision Tree: 0.9121
```

In [99]:

```
fpr_tree, tpr_tree, thresholds_tree = roc_curve(y_test, tree.predict_proba(X_testp)[:,1
])
plt.plot(fpr_tree, tpr_tree, label = 'Decision Tree')
plt.plot(fpr_xgb, tpr_xgb, label = 'XGBoost')
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.title("ROC Curve for Tree and XGBoost")
plt.legend()
plt.show()
```




Based on test score and ROC curve, our tree model is obviously doing a little bit worse than the XGBoost, but still doing very well compared to, for instance, our baseline model (its test score was 0.84). We also tried building the tree with 8 max leaf nodes, which made it more "explainable", but its score decreased a lot so we decided to keep 16 max leaf nodes. Let's now try to visualize the tree to explain the decisions it takes:

In [98]:

```
feature_names = prepnsc.named_transformers_['one-hot encoding'].get_feature_names().tol
ist()
feature_names.extend(X_train.columns[ord_mask])
feature_names.extend(X_train.columns[cont_mask])

tree_dot = export_graphviz(tree, out_file=None, feature_names=feature_names)
graph = graphviz.Source(tree_dot)
graph.render(cleanup=True)
graph
```

Out[98]: