Porto Seguro는 브라질의 자동차 보험 회사입니다.

이 캐글 대회의 목적은 어떤 차주가 내년에 보험 청구를 할 확률을 예측하는 것입니다.
Feature가 무엇을 뜻하는지 제시하지 않았다는 것이 특징이다. 단, Feature가 binary인지, categorical인지, ordinal인지, nominal인지만 구분할 수 있고 보안상 공개하지 않았다고 한다.

target은 보험청구를 한다(1), 보험 청구를 하지 않는다(0)인 binary 데이터이다. 0과 1로 구성되어있으며 0이 1보다 압도적으로 많다.

In [1]:

```
# !pip install lightgbm
# !pip install xgboost
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.utils import shuffle
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.feature_selection import VarianceThreshold, SelectFromModel

from sklearn.model_selection import StratifiedKFold, cross_val_score

from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression

pd.set_option('display.max_columns', 100) #표를 출력할 때 최대 열 수
```

In [2]:

```
trainset = pd.read_csv('./input/train.csv')
testset = pd.read_csv('./input/test.csv')
```

In [3]:

```
trainset.head()
```

Out[3]:

	id	target	ps_ind_01	ps_ind_02_cat	ps_ind_03	ps_ind_04_cat	ps_ind_05_cat	ps_ind_06_b
0	7	0	2	2	5	1	0	
1	9	0	1	1	7	0	0	
2	13	0	5	4	9	1	0	
3	16	0	0	1	2	0	0	
4	17	0	0	2	0	1	0	

- 비슷한 Group에 있는 feature들은 비슷한 이름을 갖고 있다.
- bin : Binary Feature (0 또는 1의 값을 가짐)

- cat : Categorical Feature (각각의 번호(정수)가 부여된다)
- 이외의 feature들은 Continuous 혹은 Ordinal feature이다.
- 값이 -1인 것은 결측지(NaN)을 의미한다.
- Target column은 policy holder에게 청구 적용 여부 (Y/N)을 의미한다.

In [4]:

```
print("Train dataset (rows, cols):", trainset.shape, "₩nTest dataset (rows, cols):", testset.shape)
```

```
Train dataset (rows, cols): (595212, 59)
Test dataset (rows, cols): (616098, 58)
```

Train dataset에는 59개의 column이 있고 Test dataset에는 58개의 column이 있는데 그 이유는 Test dataset에서 target column이 빠졌기 때문이다.

In [5]:

```
print("Columns in train and not in test dataset : ", set(trainset.columns)-set(testset.columns))
```

```
Columns in train and not in test dataset :  {'target'}
```

In [6]:

```
trainset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 595212 entries, 0 to 595211
Data columns (total 59 columns):
id                595212 non-null int64
target            595212 non-null int64
ps_ind_01         595212 non-null int64
ps_ind_02_cat     595212 non-null int64
ps_ind_03         595212 non-null int64
ps_ind_04_cat     595212 non-null int64
ps_ind_05_cat     595212 non-null int64
ps_ind_06_bin     595212 non-null int64
ps_ind_07_bin     595212 non-null int64
ps_ind_08_bin     595212 non-null int64
ps_ind_09_bin     595212 non-null int64
ps_ind_10_bin     595212 non-null int64
ps_ind_11_bin     595212 non-null int64
ps_ind_12_bin     595212 non-null int64
ps_ind_13_bin     595212 non-null int64
ps_ind_14         595212 non-null int64
ps_ind_15         595212 non-null int64
ps_ind_16_bin     595212 non-null int64
ps_ind_17_bin     595212 non-null int64
ps_ind_18_bin     595212 non-null int64
ps_reg_01         595212 non-null float64
ps_reg_02         595212 non-null float64
ps_reg_03         595212 non-null float64
ps_car_01_cat     595212 non-null int64
ps_car_02_cat     595212 non-null int64
ps_car_03_cat     595212 non-null int64
ps_car_04_cat     595212 non-null int64
ps_car_05_cat     595212 non-null int64
ps_car_06_cat     595212 non-null int64
ps_car_07_cat     595212 non-null int64
ps_car_08_cat     595212 non-null int64
ps_car_09_cat     595212 non-null int64
ps_car_10_cat     595212 non-null int64
ps_car_11_cat     595212 non-null int64
ps_car_11         595212 non-null int64
ps_car_12         595212 non-null float64
ps_car_13         595212 non-null float64
ps_car_14         595212 non-null float64
ps_car_15         595212 non-null float64
ps_calc_01        595212 non-null float64
ps_calc_02        595212 non-null float64
ps_calc_03        595212 non-null float64
ps_calc_04        595212 non-null int64
ps_calc_05        595212 non-null int64
ps_calc_06        595212 non-null int64
ps_calc_07        595212 non-null int64
ps_calc_08        595212 non-null int64
ps_calc_09        595212 non-null int64
ps_calc_10        595212 non-null int64
ps_calc_11        595212 non-null int64
ps_calc_12        595212 non-null int64
ps_calc_13        595212 non-null int64
ps_calc_14        595212 non-null int64
ps_calc_15_bin    595212 non-null int64
```

```
ps_calc_16_bin    595212 non-null int64
ps_calc_17_bin    595212 non-null int64
ps_calc_18_bin    595212 non-null int64
ps_calc_19_bin    595212 non-null int64
ps_calc_20_bin    595212 non-null int64
dtypes: float64(10), int64(49)
memory usage: 267.9 MB
```

- 데이터들의 타입이 int64 혹은 float64 로 이루어져 있음을 확인.
- info() 메소드 상으로는 결측값이 없는 것으로 확인됨. 왜냐하면 결측값이 -1로 대체되어있기 때문.

Metadata에 관하여

나중에 데이터 분석 시 용이하게 사용하기 위해 메타데이터의 일부 내용을 trainset의 변수와 연관지어서 분석할 예정.

사용할 메타데이터

- use : input, ID, target
- type : nominal, interval, ordinal, binary
- preserve : True or False
- dataType : int, float, char
- category, ind, reg, car, calc

In [7]:

```
data = []
for feature in trainset.columns:
    if feature == 'target':
        use = 'target'
    elif feature == 'id':
        use = 'id'
    else:
        use = 'input'

    # Type 정의
    if 'bin' in feature or feature == 'target':
        type = 'binary'
    elif 'cat' in feature or feature == 'id':
        type = 'categorical'
    elif trainset[feature].dtype == float or isinstance(trainset[feature].dtype, float):
        type = 'real'
    elif trainset[feature].dtype == int:
        type = 'integer'
    #isinstance() : 자료형을 확인하는 함수

    # preserve변수를 id 제외하고 모든 변수들에 대해서는 True로 초기화
    #preserve가 True이면 해당 데이터를 활용하지않고 보존, False면 활용
    preserve = True
    if feature == 'id':
        preserve = False

    # Datatype 정의
    dtype = trainset[feature].dtype

    category = 'none'
    # Category 정의
    if 'ind' in feature:
        category = 'individual'
    elif 'reg' in feature :
        category = 'registration'
    elif 'car' in feature :
        category = 'car'
    elif 'calc' in feature :
        category = 'calculated'

    # 모든 변수에 대한 metadata를 포함하고 있는 dictionary 정의
    feature_dictionary = {
        'varname' : feature,
        'use':use,
        'type':type,
        'preserve':preserve,
        'dtype':dtype,
        'category':category
    }
    data.append(feature_dictionary)

    metadata = pd.DataFrame(data, columns = ['varname', 'use', 'type', 'preserve', 'dtype', 'categor
    metadata.set_index('varname', inplace = True)
    print(metadata)
```

```
        use         type  preserve  dtype category
varname
id       id  categorical     False  int64     none
```

```
            use         type  preserve  dtype category
varname
id           id  categorical     False  int64     none
target   target       binary      True  int64     none
                use         type  preserve  dtype    category
varname
id              id  categorical     False  int64        none
target      target       binary      True  int64        none
ps_ind_01    input      integer      True  int64  individual
                    use         type  preserve  dtype    category
varname
id                   id  categorical     False  int64        none
target           target       binary      True  int64        none
ps_ind_01         input      integer      True  int64  individual
ps_ind_02_cat     input  categorical      True  int64  individual
                    use         type  preserve  dtype    category
```

In [8]:

```
pd.DataFrame({'count':metadata.groupby(['category'])['category'].size()}).reset_index()
```

Out[8]:

	category	count
0	calculated	20
1	car	16
2	individual	18
3	none	2
4	registration	3

각 카테고리별 개수

In [9]:

```
pd.DataFrame({'count':metadata.groupby(['use','type'])['use'].size()}).reset_index()
```

Out[9]:

	use	type	count
0	id	categorical	1
1	input	binary	17
2	input	categorical	14
3	input	integer	16
4	input	real	10
5	target	binary	1

use와 type에 따른 target 수

Data analysis and statistics

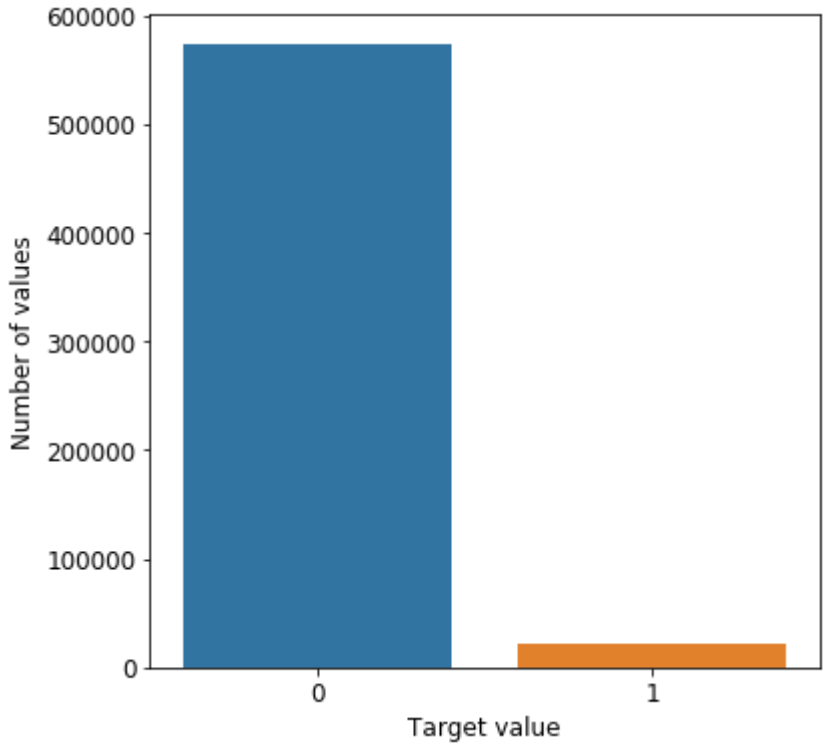
Target variable

In [10]:

```
plt.figure()
fig,ax = plt.subplots(figsize = (6,6))
x = trainset['target'].value_counts().index.values #0인지 1인지를 나타내는 값
y = trainset['target'].value_counts().values #target value를 count

# Bar plot
sns.barplot(ax = ax, x = x, y = y)
plt.ylabel('Number of values', fontsize = 12)
plt.xlabel('Target value', fontsize = 12)
plt.tick_params(axis = 'both', which = 'major', labelsize = 12) #axis = both : x,y축의 틱(눈금)이 도
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



겨우 3.64%의 target data만 1이라는 값을 가지고 있다. 이는 훈련 데이터셋이 매우 불균형함을 의미한다. 사실 0이 1보다 압도적으로 많으면 모든 target 값을 0으로 예측해도 정확도가 높다. 하지만 그건 아무런 가치가 없는 예측이고 우리가 원하는 건 1을 얼마나 잘 예측하느냐 이다.

이 불균현 문제를 해결하기 위해 2가지 방법을 사용할 수 있는데

- oversampling records with target = 1
- undersampling records with target = 0 우리가 사용하는 training set의 규모가 크니까 undersampling을 진행한다.

Undersampling vs Oversampling

출처 : https://hwiyong.tistory.com/266 (https://hwiyong.tistory.com/266)

Undersampling (과소표집)

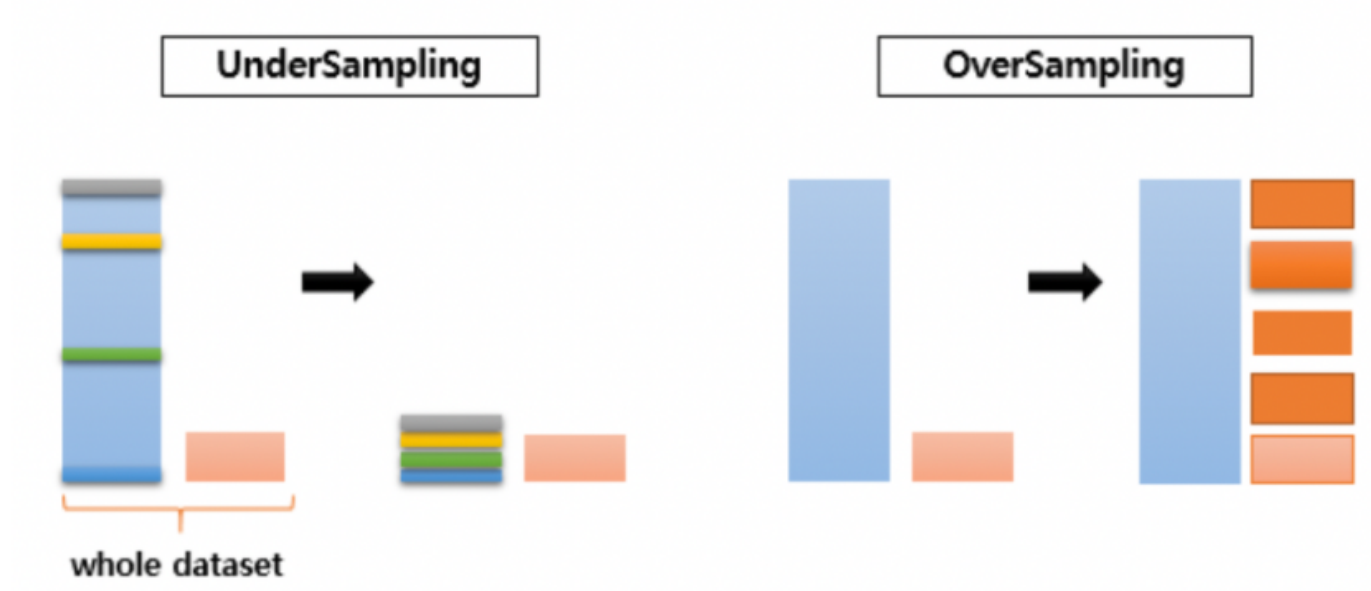
Undersampling (과소표집)

과소표집은 다른 클래스에 비해 상대적으로 많이 나타나 있는 클래스의 개수를 줄이는 것. 균형을 유지할 수 있게 되지만, 제거하는 과정에서 유용한 정보가 버려지게 되는 것이 큰 단점.

Oversampling (과대표집)

과대표집은 데이터를 복제하는 것. 무작위로 하는 경우도 있고, 기준을 미리 정해서 복제하는 방법도 있다. 정보를 잃지 않고 , 훈련용 데이터에서 높은 성능을 보이지만 실험용 데이터에서의 성능은 낮아질 수 있다. 대부분의 과대 표집 방법은 Overfitting 문제를 포함하고 있어서 이를 피하기 위해 SMOTE(Syntethic Minority Over-sampling Technique)를 사용한다고 한다.

SMOTE : 데이터의 개수가 적은 클래스의 표본(Sample)을 가져온 뒤에 임의의 값을 추가하여 새로운 샘플을 만들어 데이터에 추가. 이 과정에서 각 표본은 주변 데이터를 고려하기 때문에 과대적합의 가능성이 낮아진다.



Real features

In [11]:

```
variable = metadata[(metadata.type == 'real')&(metadata.preserve)].index
trainset[variable].describe()
```

Out[11]:

	ps_reg_01	ps_reg_02	ps_reg_03	ps_car_12	ps_car_13	ps_car_
count	595212.000000	595212.000000	595212.000000	595212.000000	595212.000000	595212.000(
mean	0.610991	0.439184	0.551102	0.379945	0.813265	0.276:
std	0.287643	0.404264	0.793506	0.058327	0.224588	0.357
min	0.000000	0.000000	-1.000000	-1.000000	0.250619	-1.000(
25%	0.400000	0.200000	0.525000	0.316228	0.670867	0.333
50%	0.700000	0.300000	0.720677	0.374166	0.765811	0.368
75%	0.900000	0.600000	1.000000	0.400000	0.906190	0.396
max	0.900000	1.800000	4.037945	1.264911	3.720626	0.636

Real feature에 해당하는 variable들의 특징을 파악

- reg variables : ps_reg_03의 min값을 보면 결측치를 갖고 있음을 알 수 있다.
- car variables ; ps_car_12, ps_car_14가 결측치를 갖고 있다.
- calc variables : 결측치가 없다. 최대값이 0.9인것을 보아 일종의 비율을 나타낸 값으로 추측된다.

In [14]:

```
(pow(trainset['ps_car_12']*10,2)).head(10)
```

Out[14]:

```
0    16.00
1    10.00
2    10.00
3    14.00
4     9.99
5    19.89
6    10.00
7    19.98
8    16.00
9    20.00
Name: ps_car_12, dtype: float64
```

In [13]:

```
(pow(trainset['ps_car_15'],2)).head(10)
```

Out[13]:

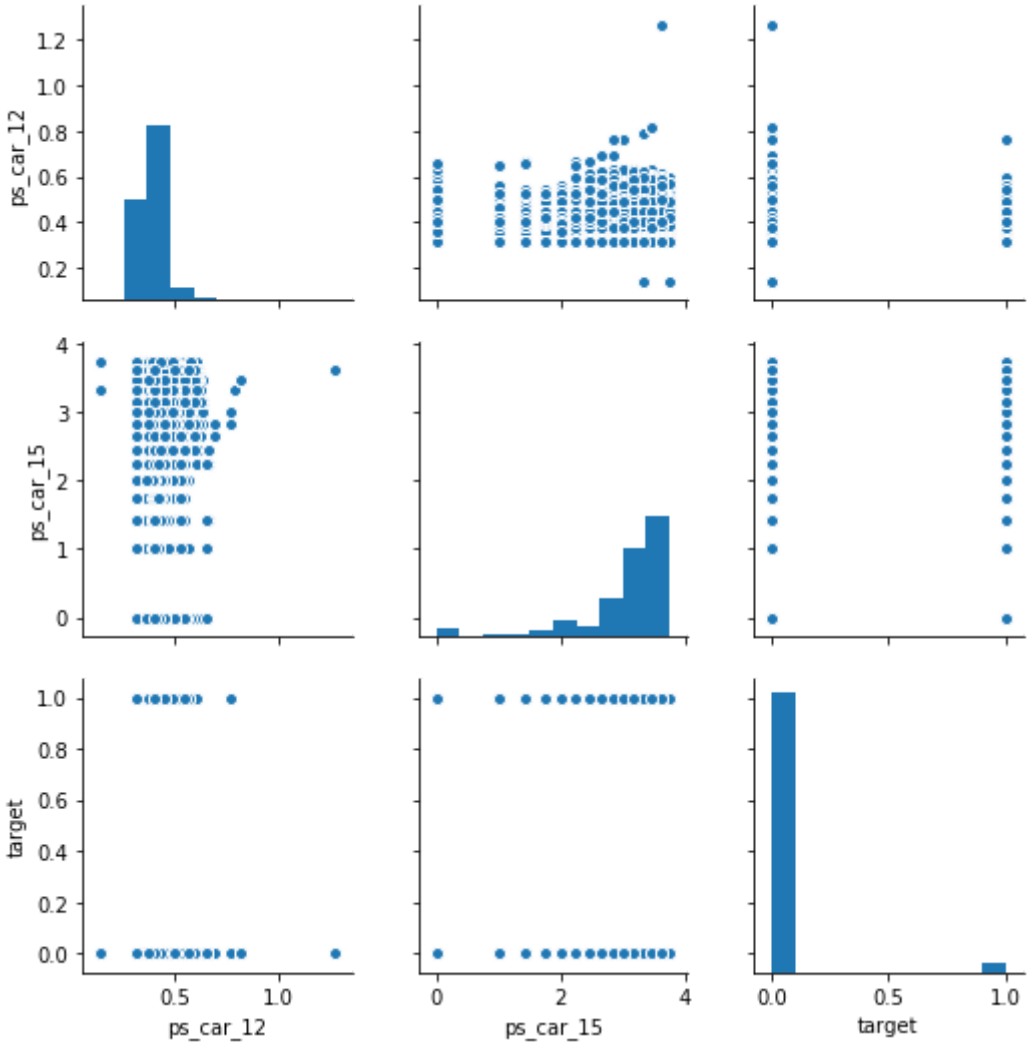
```
0    13.0
1     6.0
2    11.0
3     4.0
4     4.0
5     9.0
6    10.0
7    11.0
8     8.0
9    13.0
Name: ps_car_15, dtype: float64
```

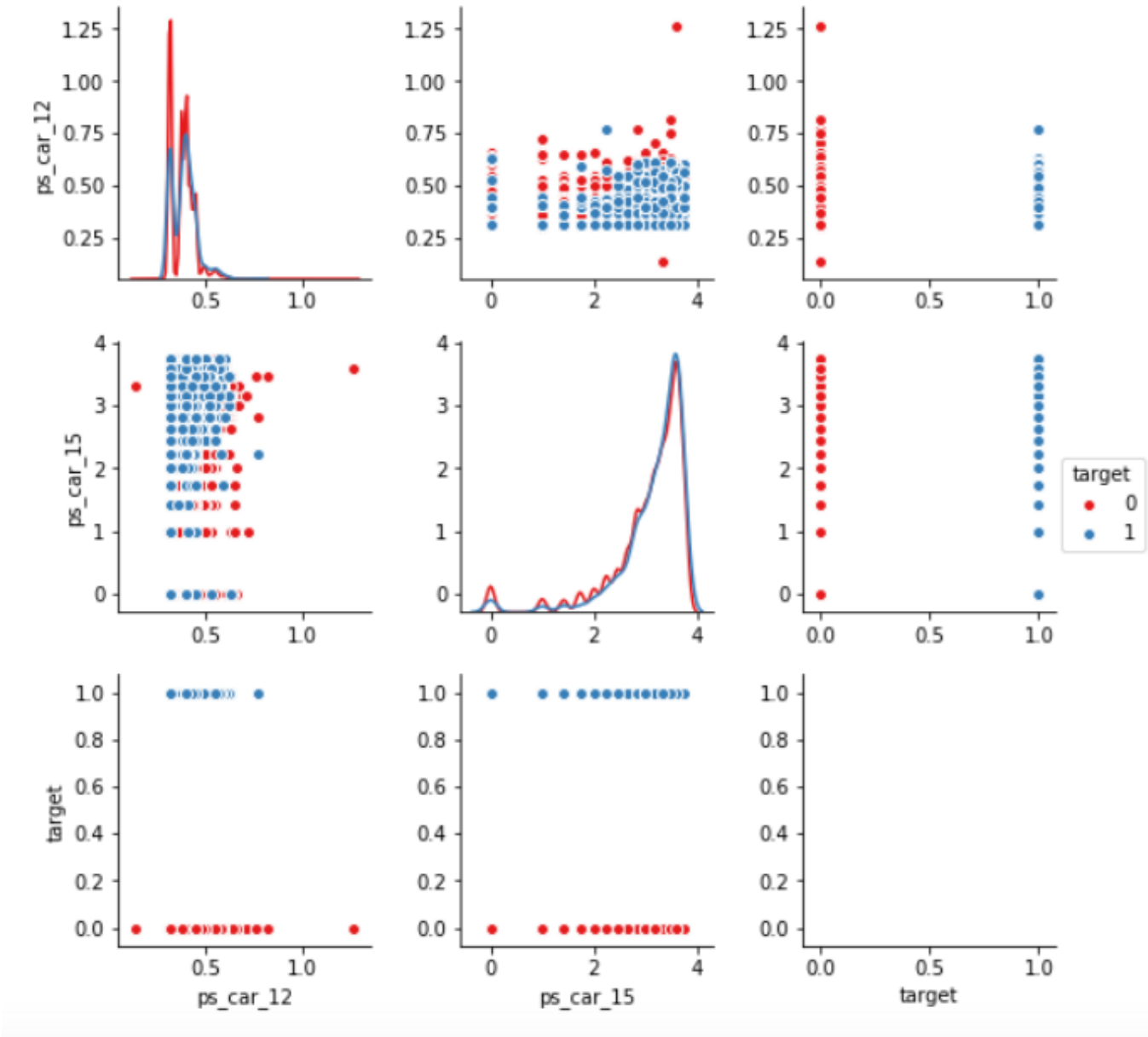
Car feature

ps_car_12를 10으로 나눈것과 ps_car_15은 자연수의 제곱근을 의미한다. 이러한 값들은 pairplot을 이용해서 표시해줄 수 있다고 한다.

In [23]:

```
sample = trainset.sample(frac=0.05) # frac은 전체 개수의 비율만큼 샘플을 반환할 경우 사용
var = ['ps_car_12', 'ps_car_15', 'target']
sample = sample[var]
#######수정된 부분#######################
try:
    sns.pairplot(sample)
except RuntimeError as re:
    if str(re).startswith("Selected KDE bandwidth is 0. Cannot estimate density."):
        sns.pairplot(sample, kde_kws={'bw': 0.1}, hue = 'target', palette='Set1', diag_kind = 'kde'
    else:
        raise re
########################################
#sns.pairplot(sample,  hue='target', palette = 'Set1', diag_kind='kde')
plt.show()
```





원래는 이렇게 나와야 한다는데.. 저 코드로 돌리면 에러가 나서 수정했더니 조금 다르게 나오게 되었다ㅠㅠ

Calculated features

ps_calc_01, ps_calc_02, ps_calc_03은 매우 비슷한 분포를 갖고 있고 어떤 비율을 의미하는 변수이다. 왜냐하면 세 변수의 maximum 값이 0.9이기 때문이다.

Real feature들을 density plot으로 시각화해보자.

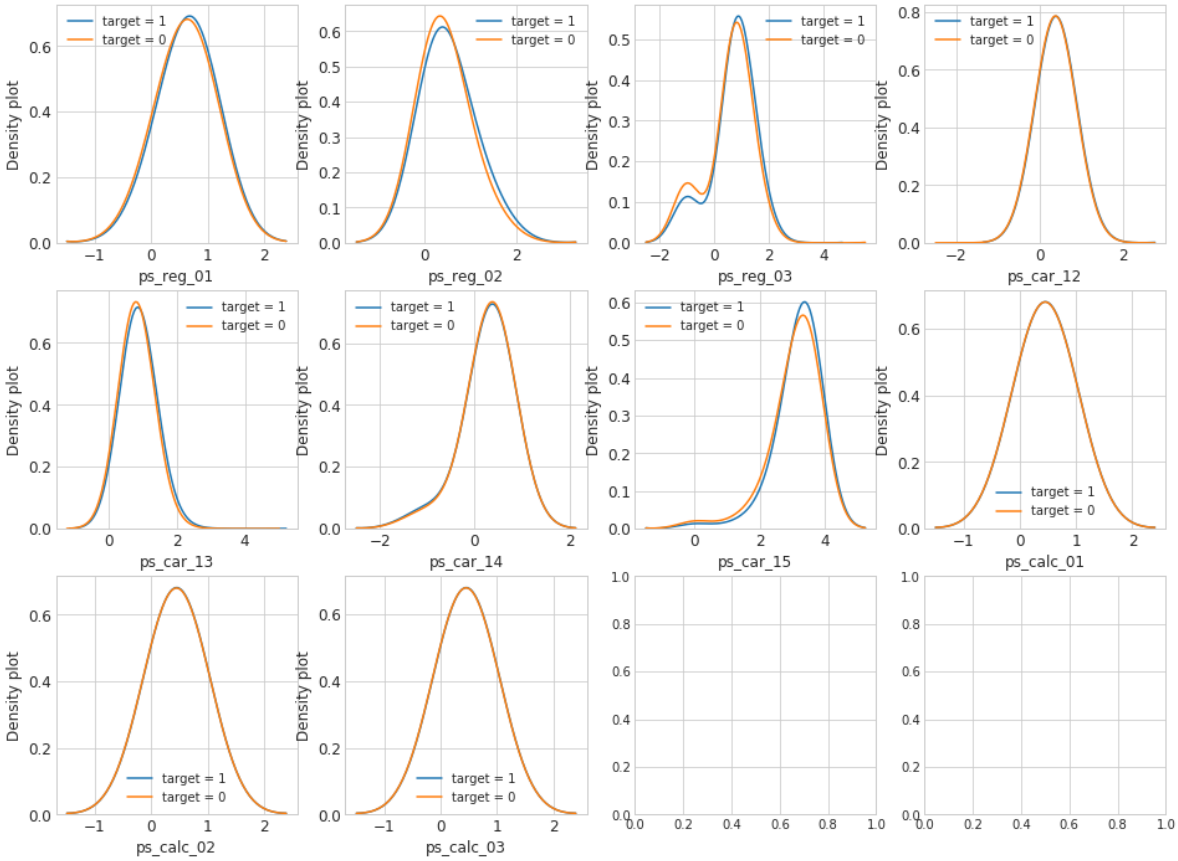
In [26]:

```
var = metadata[(metadata.type=='real')&(metadata.preserve)].index
i=0
t1 = trainset.loc[trainset['target']!=0] # 보험청구를 하는 쪽
t0 = trainset.loc[trainset['target']==0] # 보험청구를 하지 않는 쪽

sns.set_style('whitegrid')
plt.figure()
fig, ax = plt.subplots(3, 4, figsize=(16,12))

for feature in var:
    i+=1
    plt.subplot(3, 4, i)
    sns.kdeplot(t1[feature], bw = 0.5, label = 'target = 1')
    sns.kdeplot(t0[feature], bw = 0.5, label = 'target = 0')
    plt.ylabel('Density plot', fontsize = 12)
    plt.xlabel(feature, fontsize = 12)
    locs, labels = plt.xticks()
    plt.tick_params(axis = 'both', which = 'major', labelsize = 12)
plt.show()
```

<Figure size 432x288 with 0 Axes>

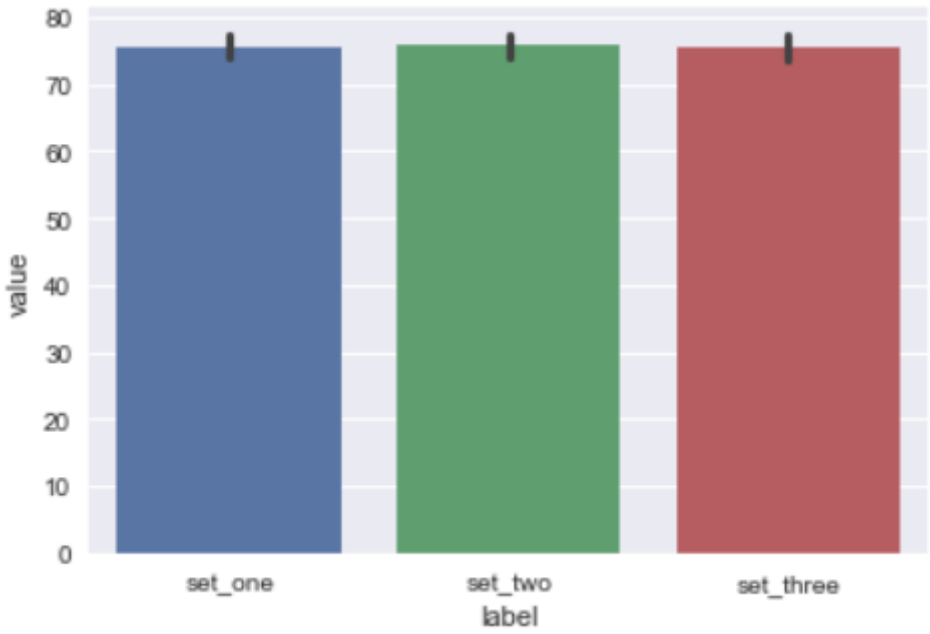


ps_reg_02, ps_car_13, ps_car_15가 target=0, target=1 사이의 분포가 차이가 많이 나는 feature라고 한다.

KDE plot에 대해 알아보자

출처 : http://hleecaster.com/python-seaborn-kdeplot/ (http://hleecaster.com/python-seaborn-kdeplot/)

막대그래프의 문제점



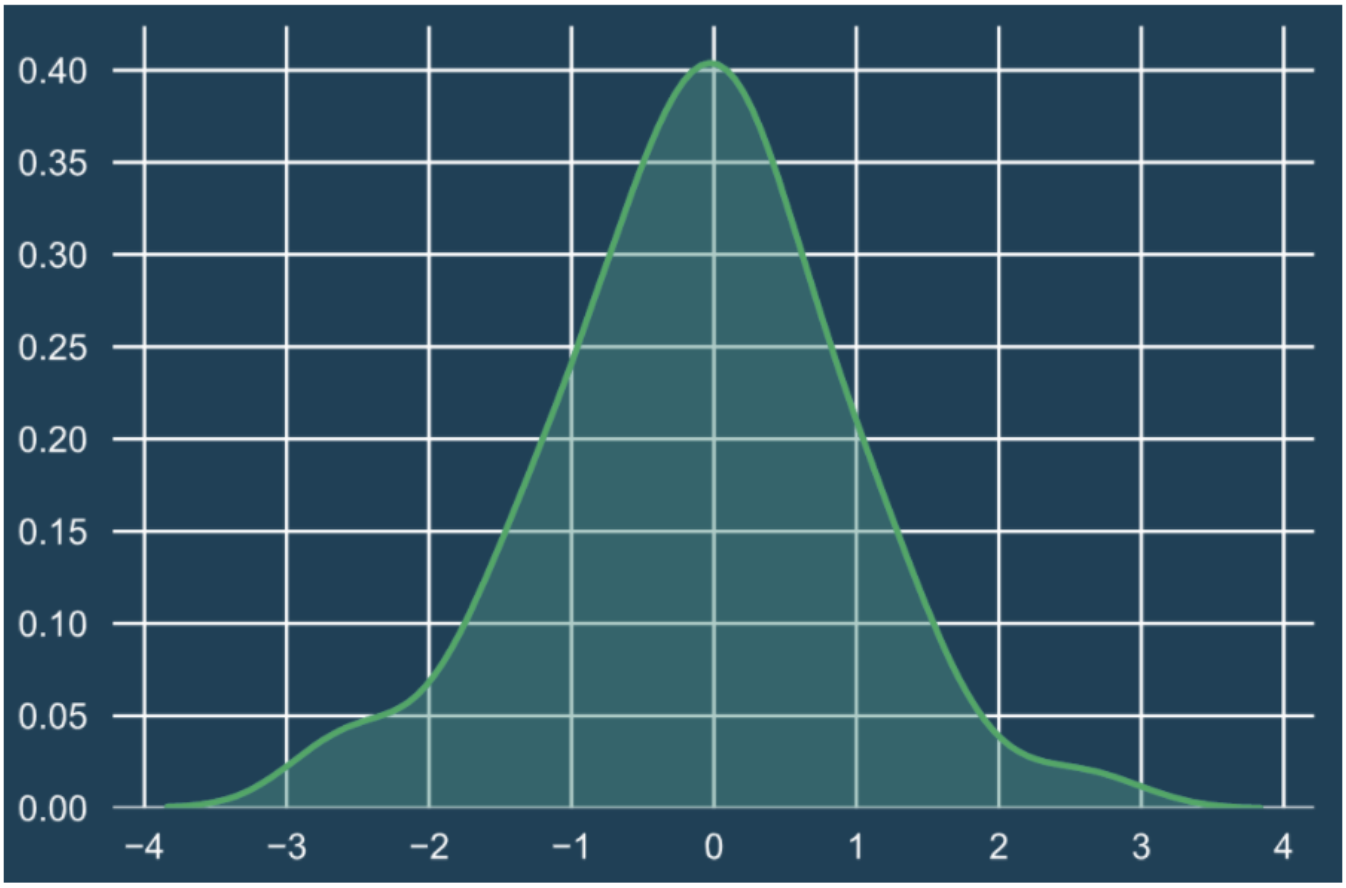
막대그래프의 문제점은 위 그림처럼 평균과 신뢰구간이 비슷한 경우, 어떤 데이터인지 구분이 안된다. 최대값, 최소값은 어떤지, 어떤 모양으로 얼마나 퍼져 있는지 분포가 확인이 되지 않는다.

그래서 보통 분포를 확인할 때 히스토그램(histogram)을 많이 활용하곤 한다. 구간을 나눠서 막대 그래프를 그리는 건데 히스토그램은 막대 하나의 bin 구간을 어떻게 설정하느냐에 따라 결과물이 매우 달라져서 엉뚱한 결론과 해석을 내릴 수 있어서 조심해야한다.

그래서 그 대안으로 많이 쓰이는 게 커널 밀도 추정(KDE : Kernel Density Estimator)이다.

커널 밀도 추정(KDE : Kernel Density Estimator)이란

쉽게 얘기하면 히스토그램 같은 분포를 스무딩, 부드럽게 곡선화 시켜서 그려주는 것이다.



자세히는 여기1 (https://darkpgmr.tistory.com/147)랑 여기2 (https://niceguy1575.tistory.com/entry/Kernel-Density-EstimationKDE-%EC%BB%A4%EB%84%90%EB%B0%80%EB%8F%84%ED%95%A8%EC%88%98)서 참고

Real feature간의 correlation에 대해 알아보자.

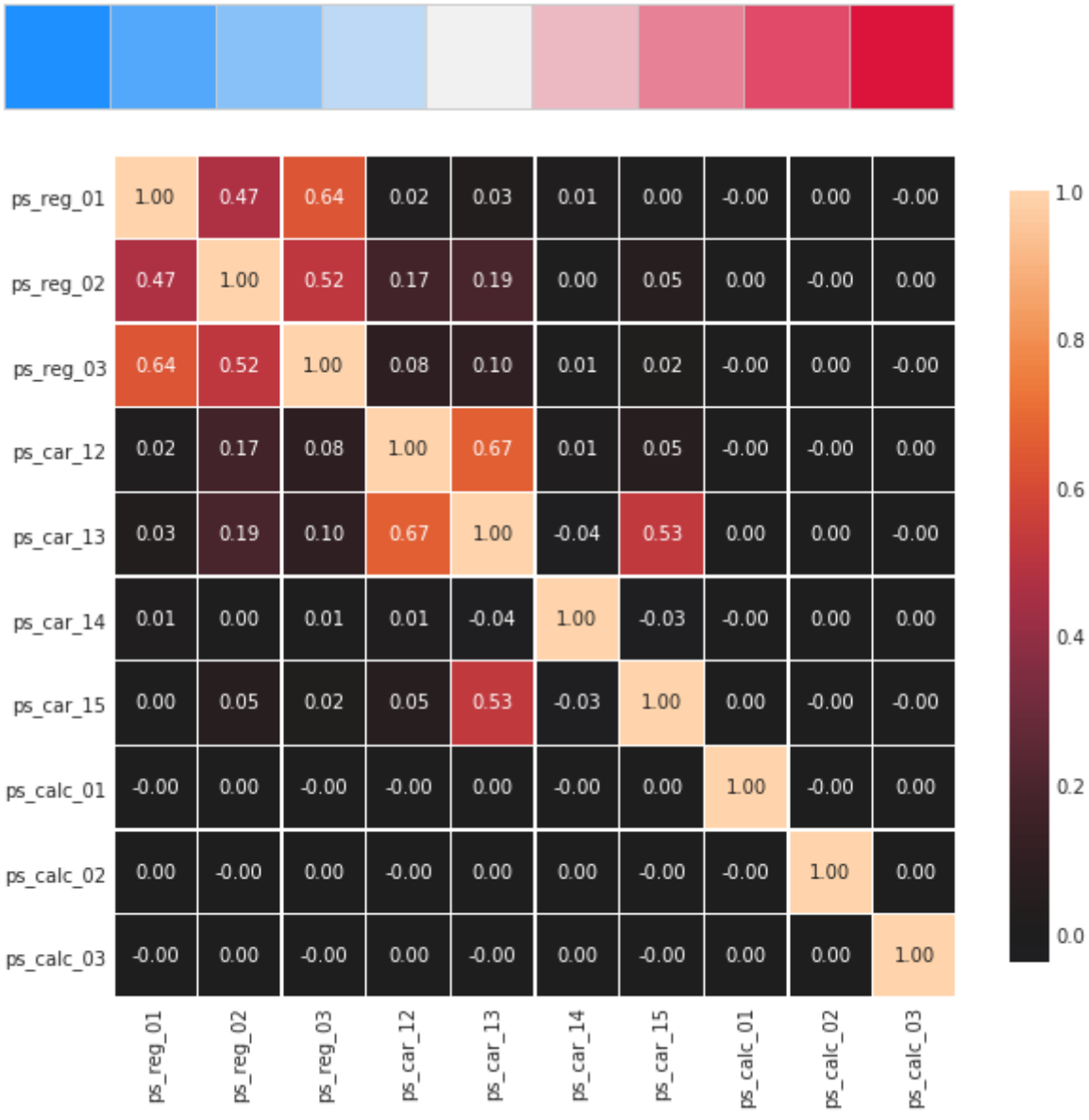
In [40]:

```
#!pip install seaborn==0.10.0
#!pip install numpy==1.17.4
def corr_heatmap(var):
    correlations = trainset[var].corr()


#########코드 수정#################
    # Create color map ranging between two colors
    cmap = sns.palplot(sns.blend_palette(["dodgerblue", ".95", "crimson"],9))
    #cmap = sns.diverging_palette(220, 10, as_cmap=True)
###################################

    fig, ax = plt.subplots(figsize=(10,10))
    sns.heatmap(correlations, cmap=cmap, vmax=1.0, center=0, fmt='.2f',
                square=True, linewidths=.5, annot=True, cbar_kws={"shrink": .75})
    plt.show();

var = metadata[(metadata.type == 'real') & (metadata.preserve)].index
corr_heatmap(var)
print(var)
```



```
Index(['ps_reg_01', 'ps_reg_02', 'ps_reg_03', 'ps_car_12', 'ps_car_13',
       'ps_car_14', 'ps_car_15', 'ps_calc_01', 'ps_calc_02', 'ps_calc_03'],
      dtype='object', name='varname')
```

각 feature들의 correlation

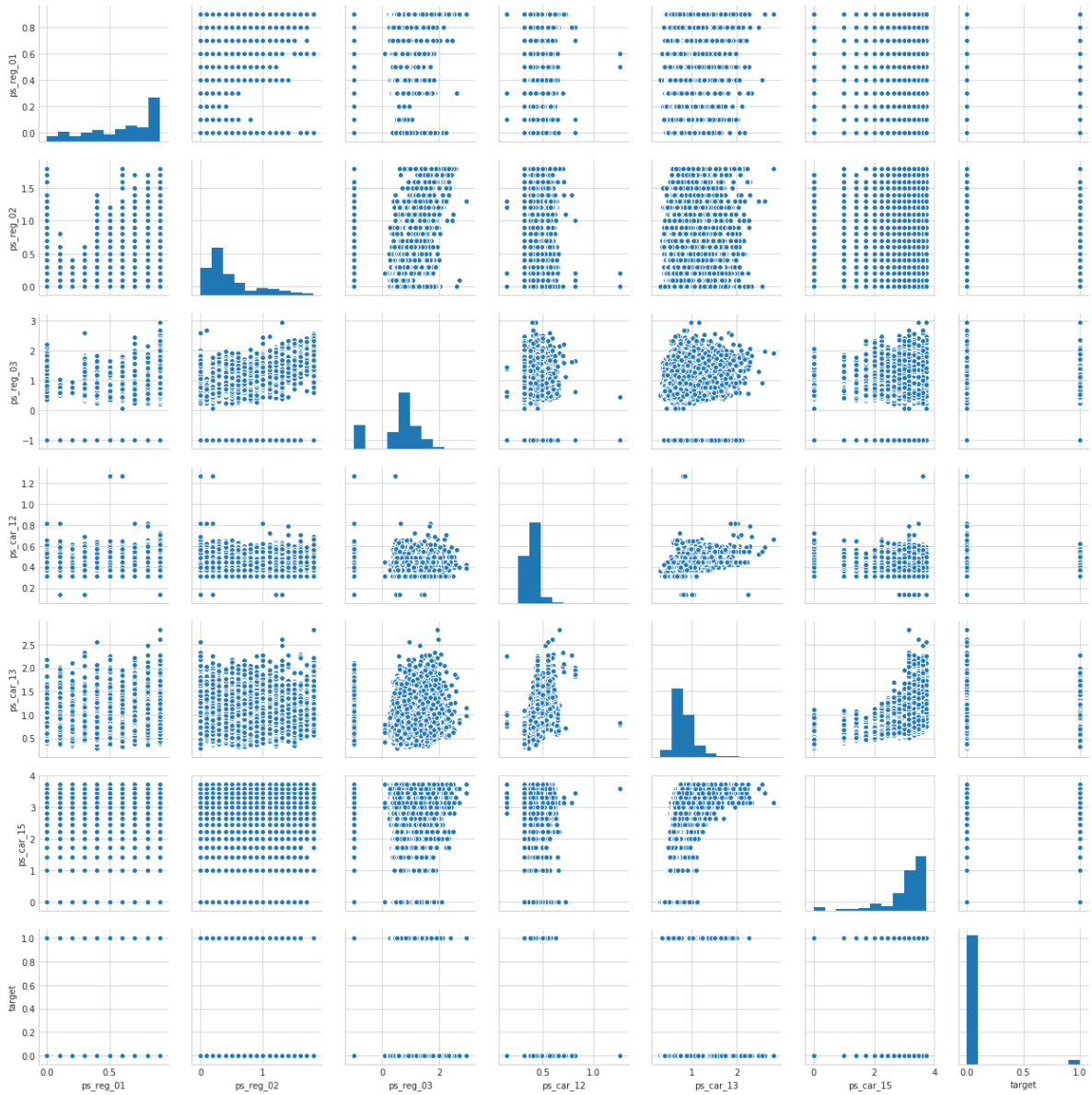
- ps_reg_01 과 ps_reg_02 (0.47)
- ps_reg_01 과 ps_reg_03 (0.64)
- ps_reg_02 과 ps_reg_03 (0.52)
- ps_car_12 과 ps_car_13 (0.67)
- ps_car_13 과 ps_car_15 (0.53)

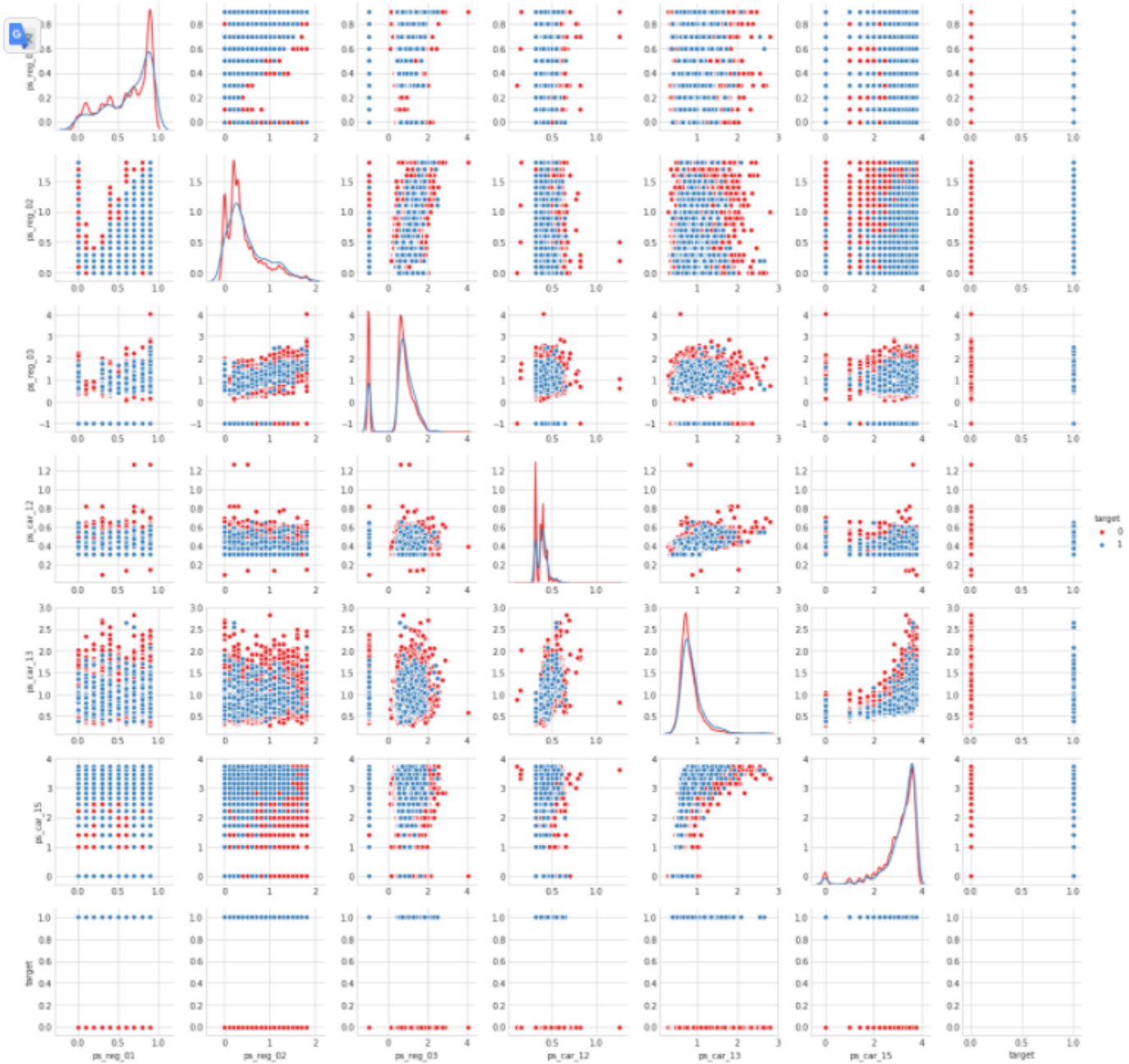
각각의 value가 correlated 되어있다는 것을 보여주기 위해 pairplot을 이용한다고 한다. 각 pair를 나타내기 전에 우리는 sample의 2%만을 뽑아서 data를 subsample 해보자.

- pairplot : 데이터에 들어있는 각 컬럼들의 상관관계를 출력할 때 사용

In [42]:

```
sample = trainset.sample(frac=0.05) # 2%만 뽑아서 subsampling
var = ['ps_reg_01', 'ps_reg_02', 'ps_reg_03', 'ps_car_12', 'ps_car_13', 'ps_car_15', 'target']
sample = sample[var]
try:
    sns.pairplot(sample)
except RuntimeError as re:
    if str(re).startswith("Selected KDE bandwidth is 0. Cannot estimate density."):
        sns.pairplot(sample, kde_kws={'bw': 0.1}, hue = 'target', palette='Set1', diag_kind = 'kde'
    else:
        raise re
# sns.pairplot(sample,  hue='target', palette = 'Set1', diag_kind='kde')
plt.show()
```





Binary Features

In [45]:

```
v = metadata[(metadata.type == 'binary') & (metadata.preserve)].index
trainset[v].describe()
```

Out[45]:

	target	ps_ind_06_bin	ps_ind_07_bin	ps_ind_08_bin	ps_ind_09_bin	ps_ind_10_
count	595212.000000	595212.000000	595212.000000	595212.000000	595212.000000	595212.000(
mean	0.036448	0.393742	0.257033	0.163921	0.185304	0.000:
std	0.187401	0.488579	0.436998	0.370205	0.388544	0.019:
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000(
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000(
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000(
75%	0.000000	1.000000	1.000000	0.000000	0.000000	0.000(
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000(

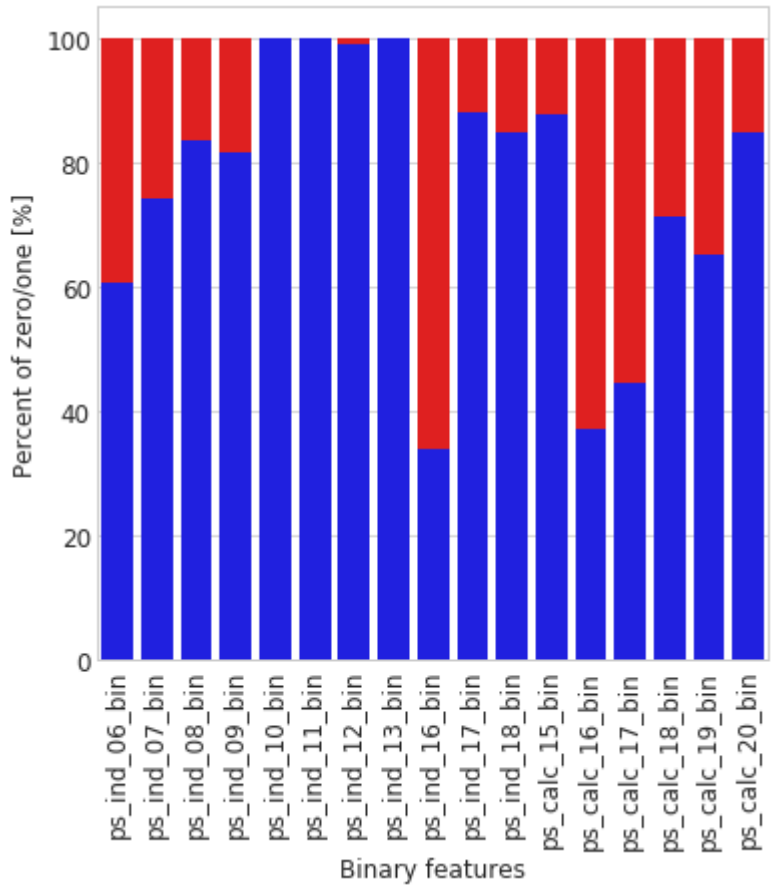
Training dataset에 있는 binary dataset의 분포를 시각화해봅시다. 파란색은 0, 빨간색은 1을 의미

In [46]:

```
bin_col = [col for col in trainset.columns if '_bin' in col]
zero_list = []
one_list = []
for col in bin_col:
    zero_list.append((trainset[col]==0).sum()/trainset.shape[0]*100)
    one_list.append((trainset[col]==1).sum()/trainset.shape[0]*100)
plt.figure()
fig, ax = plt.subplots(figsize=(6,6))
# Bar plot
p1 = sns.barplot(ax=ax, x=bin_col, y=zero_list, color="blue")
p2 = sns.barplot(ax=ax, x=bin_col, y=one_list, bottom= zero_list, color="red")
plt.ylabel('Percent of zero/one [%]', fontsize=12)
plt.xlabel('Binary features', fontsize=12)
locs, labels = plt.xticks()
plt.setp(labels, rotation=90)
plt.tick_params(axis='both', which='major', labelsize=12)
plt.legend((p1, p2), ('Zero', 'One'))
plt.show();
```

```
/home/ubuntu/anaconda3/envs/tensorflow2_p36/lib/python3.6/site-packages/ipykernel/__
main__.py:17: UserWarning: Legend does not support <matplotlib.axes._subplots.AxesSu
bplot object at 0x7fb0f10e8358> instances.
A proxy artist may be used instead.
See: http://matplotlib.org/users/legend_guide.html#creating-artists-specifically-for
-adding-to-the-legend-aka-proxy-artists (http://matplotlib.org/users/legend_guide.ht
ml#creating-artists-specifically-for-adding-to-the-legend-aka-proxy-artists)
```

```
<Figure size 432x288 with 0 Axes>
```



ps_ind_10_bin, ps_ind_11_bin, ps_ind_12_bin, ps_ind_13_bin은 1의 비율이 굉장히 작다. (0.5% 미만). 반면에 ps_ind_16, ps_cals_16_bin에는 1의 비율이 굉장이 높다. (60% 이상)

이제 binary data와 target variable의 관계를 알아보자.

In [47]:

```
var = metadata[(metadata.type == 'binary') & (metadata.preserve)].index
var = [col for col in trainset.columns if '_bin' in col]
i = 0
t1 = trainset.loc[trainset['target'] != 0]
t0 = trainset.loc[trainset['target'] == 0]

sns.set_style('whitegrid')
plt.figure()
fig, ax = plt.subplots(6,3,figsize=(12,24))

for feature in var:
    i += 1
    plt.subplot(6,3,i)
    sns.kdeplot(t1[feature], bw=0.5,label="target = 1")
    sns.kdeplot(t0[feature], bw=0.5,label="target = 0")
    plt.ylabel('Density plot', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    locs, labels = plt.xticks()
    plt.tick_params(axis='both', which='major', labelsize=12)
```

```
<Figure size 432x288 with 0 Axes>
```

ps_ind_06_bin, ps_ind_07_bin, ps_ind_17_bin은 1과 0에 대해 크게 불균형적인 분포를 보인다. ps_ind_08에서는 작은 불균형 모습을 보여준다. 나머지는 비슷한 density plot의 분포를 보여준다.

Categorical features

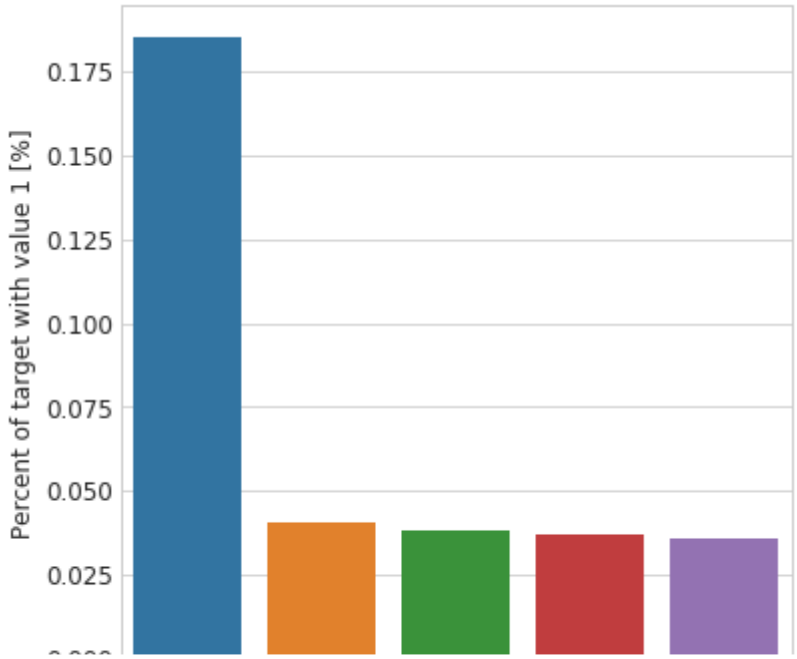
Categorical feature를 2가지 방법으로 분석해볼 예정.

1. category value별 target=1의 비율을 계산하여 bar plot으로 나타낸다.

In [49]:

```
var = metadata[(metadata.type == 'categorical') & (metadata.preserve)].index

for feature in var:
    fig, ax = plt.subplots(figsize=(6,6))
    # 카테고리 별 target=1의 퍼센티지를 구하는 방법
    cat_perc = trainset[[feature, 'target']].groupby([feature],as_index=False).mean()
    cat_perc.sort_values(by='target', ascending=False, inplace=True)
    # Bar plot
    # target mean의 내림차순으로 Bar plot을 그림
    sns.barplot(ax=ax,x=feature, y='target', data=cat_perc, order=cat_perc[feature])
    plt.ylabel('Percent of target with value 1 [%]', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    plt.tick_params(axis='both', which='major', labelsize=12)
    plt.show();
```



이번에는 target=0과 target-1인 경우를 같은 그래프 안에 나타내보자.

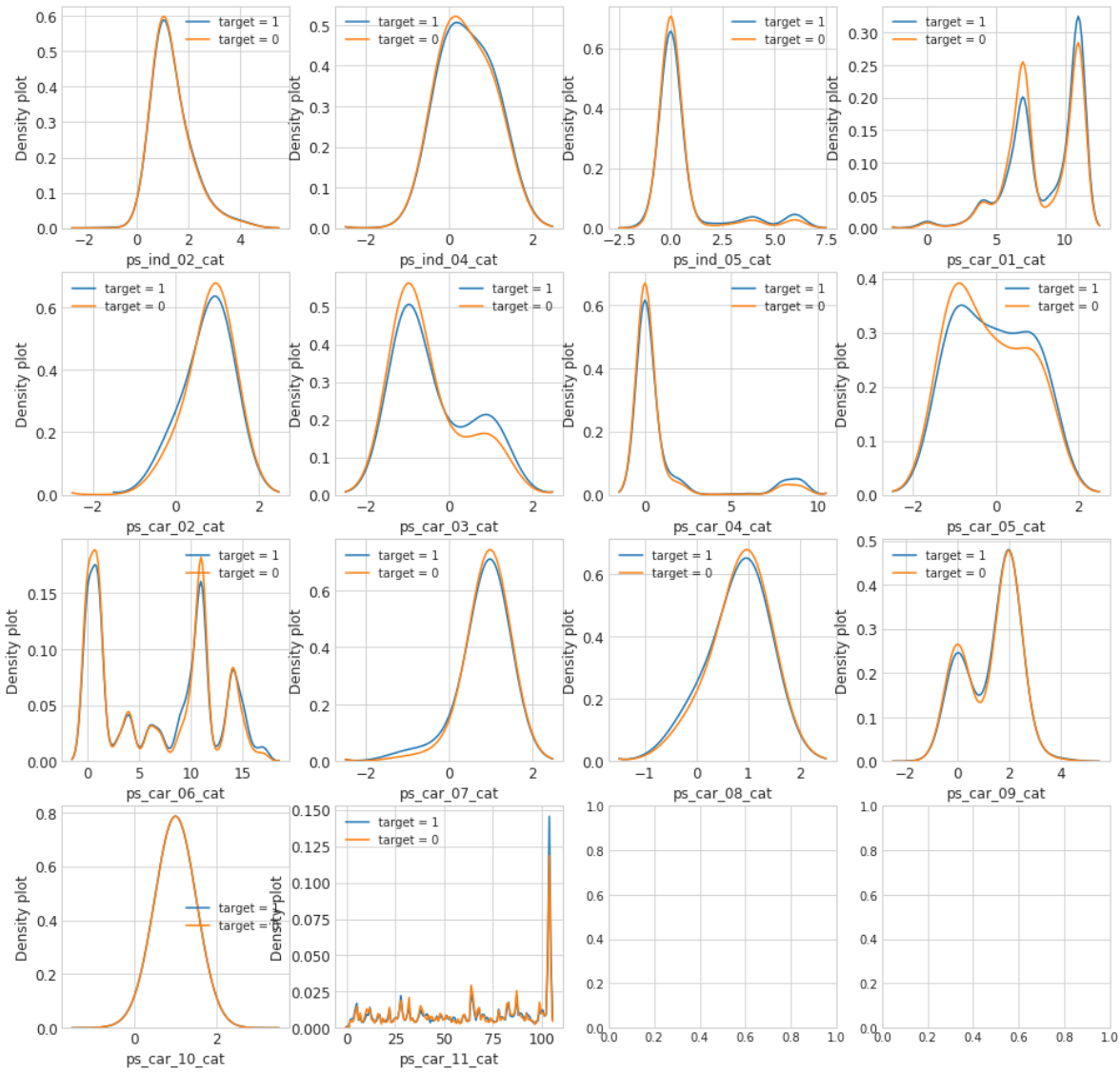
In [50]:

```
var = metadata[(metadata.type == 'categorical') & (metadata.preserve)].index
i = 0
t1 = trainset.loc[trainset['target'] != 0]
t0 = trainset.loc[trainset['target'] == 0]

sns.set_style('whitegrid')
plt.figure()
fig, ax = plt.subplots(4,4,figsize=(16,16))

for feature in var:
    i += 1
    plt.subplot(4,4,i)
    sns.kdeplot(t1[feature], bw=0.5,label="target = 1")
    sns.kdeplot(t0[feature], bw=0.5,label="target = 0")
    plt.ylabel('Density plot', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    locs, labels = plt.xticks()
    plt.tick_params(axis='both', which='major', labelsize=12)
plt.show();
```

<Figure size 432x288 with 0 Axes>



ps_car_03_cat, ps_car_05_cat은 target=0과 target=1 의 분포가 차이가 많은 value이다.

Train, Test data 사이의 data 불균형

train, test dataset의 분포에 대해 살펴보자.
먼저 reg, registration feature의 분포를 살펴보자

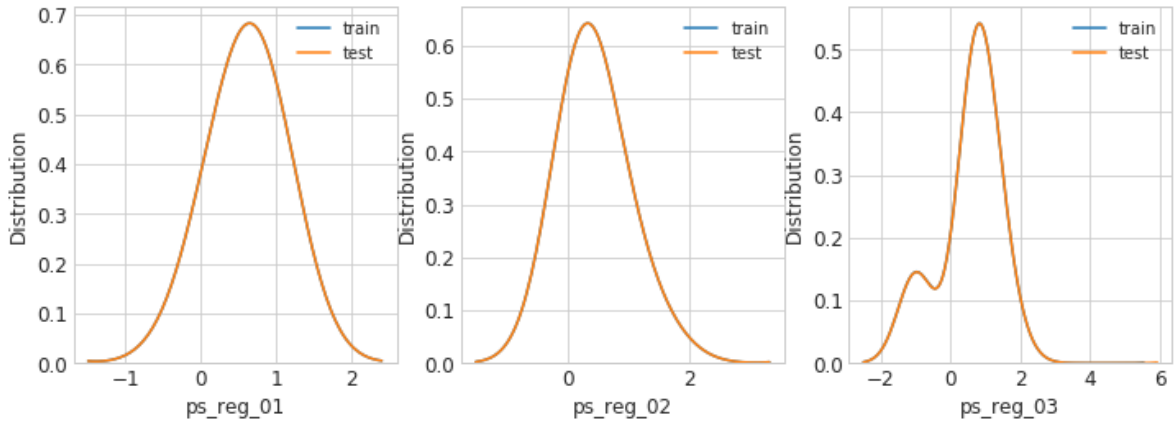
In [51]:

```
var = metadata[(metadata.category == 'registration') & (metadata.preserve)].index

# Bar plot
sns.set_style('whitegrid')

plt.figure()
fig, ax = plt.subplots(1,3,figsize=(12,4))
i = 0
for feature in var:
    i = i + 1
    plt.subplot(1,3,i)
    sns.kdeplot(trainset[feature], bw=0.5, label="train")
    sns.kdeplot(testset[feature], bw=0.5, label="test")
    plt.ylabel('Distribution', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    locs, labels = plt.xticks()
    #plt.setp(labels, rotation=90)
    plt.tick_params(axis='both', which='major', labelsize=12)
plt.show();
```

<Figure size 432x288 with 0 Axes>



모든 reg feature들을 train, test set에 대해 균형있게 분포되어있는 것을 알 수 있다. car feature에 대해 한번 살펴보자

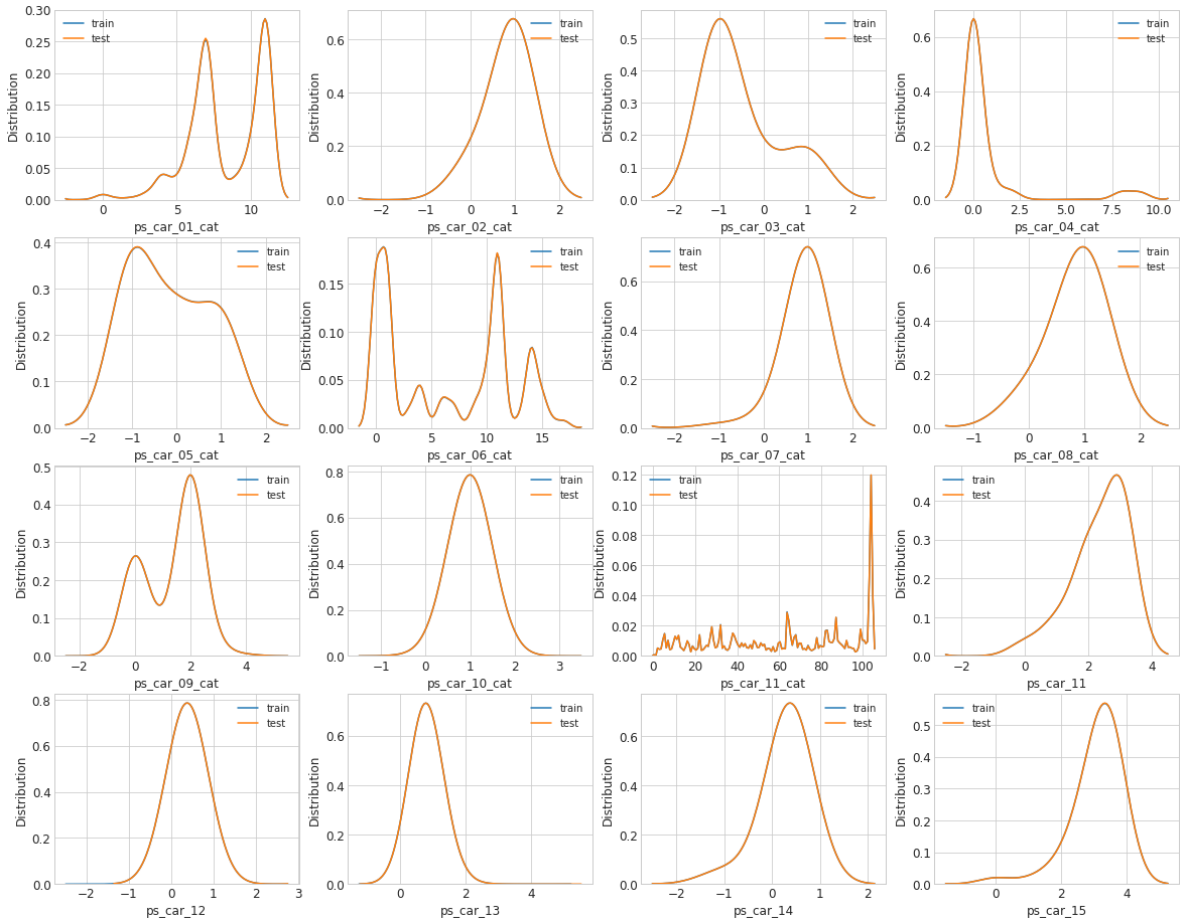
In [52]:

```
var = metadata[(metadata.category == 'car') & (metadata.preserve)].index

# Bar plot
sns.set_style('whitegrid')

plt.figure()
fig, ax = plt.subplots(4,4,figsize=(20,16))
i = 0
for feature in var:
    i = i + 1
    plt.subplot(4,4,i)
    sns.kdeplot(trainset[feature], bw=0.5, label="train")
    sns.kdeplot(testset[feature], bw=0.5, label="test")
    plt.ylabel('Distribution', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    locs, labels = plt.xticks()
    #plt.setp(labels, rotation=90)
    plt.tick_params(axis='both', which='major', labelsize=12)
plt.show();
```

<Figure size 432x288 with 0 Axes>



Car feature역시 train, test set에 대해 불균형 없이 잘 분포되어있다. individual value에 대해 살펴보자.

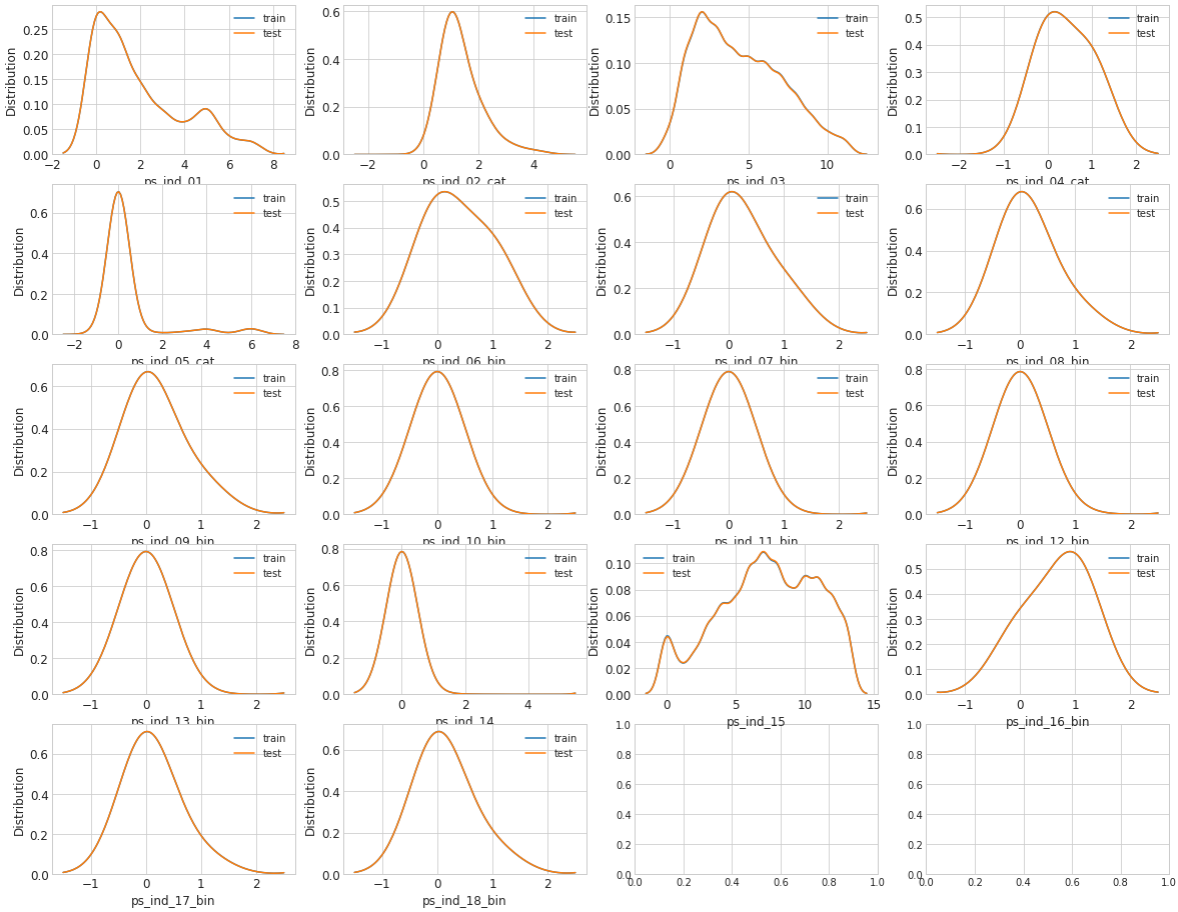
In [53]:

```
var = metadata[(metadata.category == 'individual') & (metadata.preserve)].index

# Bar plot
sns.set_style('whitegrid')

plt.figure()
fig, ax = plt.subplots(5,4,figsize=(20,16))
i = 0
for feature in var:
    i = i + 1
    plt.subplot(5,4,i)
    sns.kdeplot(trainset[feature], bw=0.5, label="train")
    sns.kdeplot(testset[feature], bw=0.5, label="test")
    plt.ylabel('Distribution', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    locs, labels = plt.xticks()
    #plt.setp(labels, rotation=90)
    plt.tick_params(axis='both', which='major', labelsize=12)
plt.show();
```

<Figure size 432x288 with 0 Axes>



ind feature도 잘 분포되어있다. calc feature에 대해 살펴보자.

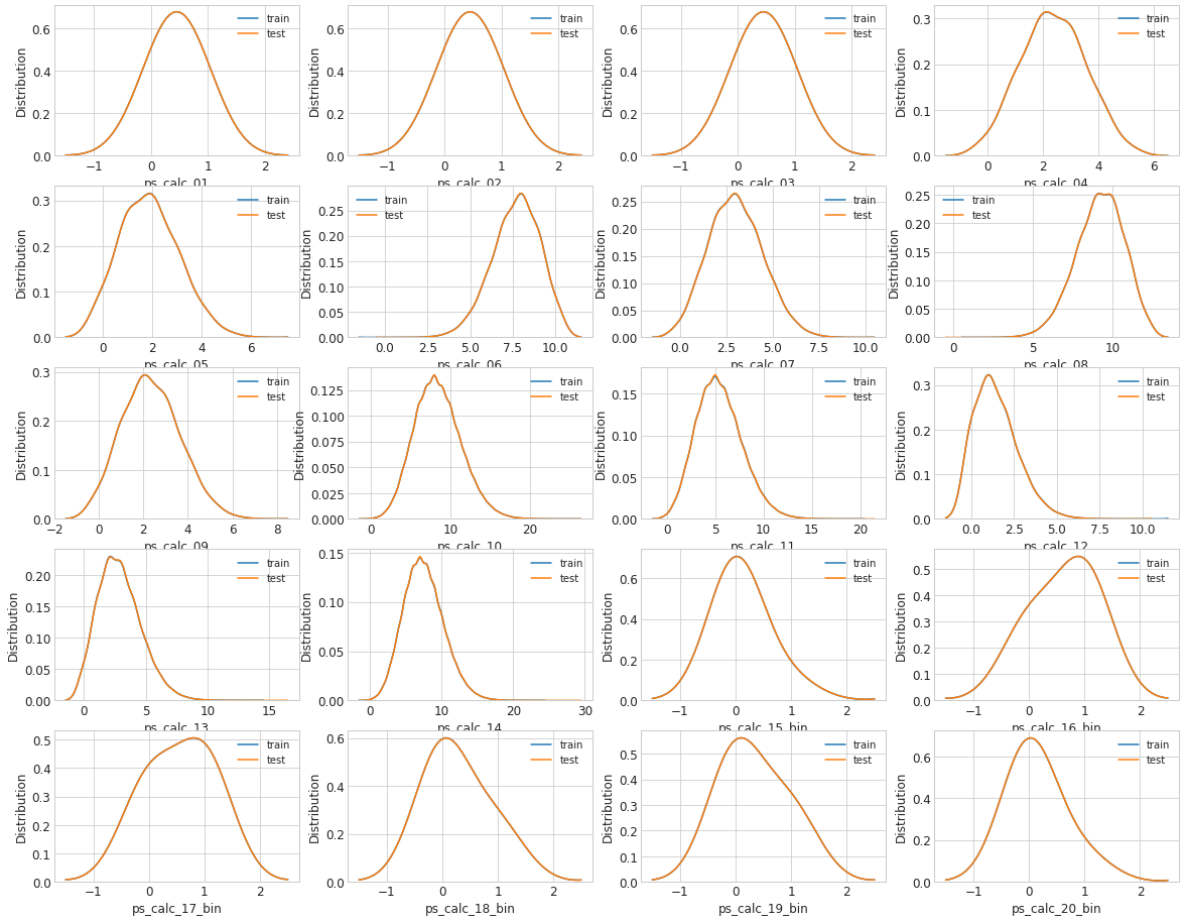
In [54]:

```
var = metadata[(metadata.category == 'calculated') & (metadata.preserve)].index

# Bar plot
sns.set_style('whitegrid')

plt.figure()
fig, ax = plt.subplots(5,4,figsize=(20,16))
i = 0
for feature in var:
    i = i + 1
    plt.subplot(5,4,i)
    sns.kdeplot(trainset[feature], bw=0.5, label="train")
    sns.kdeplot(testset[feature], bw=0.5, label="test")
    plt.ylabel('Distribution', fontsize=12)
    plt.xlabel(feature, fontsize=12)
    locs, labels = plt.xticks()
    #plt.setp(labels, rotation=90)
    plt.tick_params(axis='both', which='major', labelsize=12)
plt.show();
```

<Figure size 432x288 with 0 Axes>



calc feature도 train, test 에 잘 균형있게 분포되어있다.

(제대로 이해한건지는 모르겠지만) 일단 train, test set간의 imbalance 문제는 없어보인다. 그리고 calc feature가 모두 engineered되어서 관련이 없을 수도 있다고 한다. 이것은 하나 이상의 예측 모델을 사용하는 CV 점수를 사용해서 신중하게 연속 제거해야만 평가할 수 있다고 한다.(??)

Check data quality

Missing value에 대해 살펴봅시다

In [55]:

```
vars_with_missing = []

for feature in trainset.columns:
    missings = trainset[trainset[feature] == -1][feature].count()
    if missings > 0:
        vars_with_missing.append(feature)
        missings_perc = missings/trainset.shape[0]

        print('Variable {} has {} records ({:.2%}) with missing values'.format(feature, missings, mi

print('In total, there are {} variables with missing values'.format(len(vars_with_missing)))
```

```
Variable ps_ind_02_cat has 216 records (0.04%) with missing values
Variable ps_ind_04_cat has 83 records (0.01%) with missing values
Variable ps_ind_05_cat has 5809 records (0.98%) with missing values
Variable ps_reg_03 has 107772 records (18.11%) with missing values
Variable ps_car_01_cat has 107 records (0.02%) with missing values
Variable ps_car_02_cat has 5 records (0.00%) with missing values
Variable ps_car_03_cat has 411231 records (69.09%) with missing values
Variable ps_car_05_cat has 266551 records (44.78%) with missing values
Variable ps_car_07_cat has 11489 records (1.93%) with missing values
Variable ps_car_09_cat has 569 records (0.10%) with missing values
Variable ps_car_11 has 5 records (0.00%) with missing values
Variable ps_car_12 has 1 records (0.00%) with missing values
Variable ps_car_14 has 42620 records (7.16%) with missing values
In total, there are 13 variables with missing values
```

Prepare the data for model

본격적인 데이터 엔지니어링 시작.

Drop calc columns

앞서 말해듯이 calc column은 제거해줄 수 있다. 모두 engineered 된 것 같고 Dmitry Altukhov에 의하면 이걸 모두 지워서 CV score를 개선시킬 수 있었다고 한다. 여기(https://www.youtube.com/watch?v=mbxZ_zqHV9c (https://www.youtube.com/watch?v=mbxZ_zqHV9c)) 참고

In [56]:

```
col_to_drop = trainset.columns[trainset.columns.str.startswith('ps_calc_')]
trainset = trainset.drop(col_to_drop, axis=1)
testset = testset.drop(col_to_drop, axis=1)
```

결측치가 너무 많은 variable을 제거.

ps_car_03_cat과 ps_car_05_cat에 missing variable이 굉장히 많았음을 앞서 살펴볼 수 있었다. 이 2가지 feature 들을 제거해준다.

In [57]:

```
# Dropping the variables with too many missing values
vars_to_drop = ['ps_car_03_cat', 'ps_car_05_cat']
trainset.drop(vars_to_drop, inplace=True, axis=1)
testset.drop(vars_to_drop, inplace=True, axis=1)
metadata.loc[(vars_to_drop),'keep'] = False  # Updating the meta
```

In [58]:

```
def add_noise(series, noise_level):
    return series * (1 + noise_level * np.random.randn(len(series)))

def target_encode(trn_series=None,
                  tst_series=None,
                  target=None,
                  min_samples_leaf=1,
                  smoothing=1,
                  noise_level=0):
    """
    Smoothing is computed like in the following paper by Daniele Micci-Barreca
    https://kaggle2.blob.core.windows.net/forum-message-attachments/225952/7441/high%20cardinality%2
    trn_series : training categorical feature as a pd.Series
    tst_series : test categorical feature as a pd.Series
    target : target data as a pd.Series
    min_samples_leaf (int) : minimum samples to take category average into account
    smoothing (int) : smoothing effect to balance categorical average vs prior
    """
    assert len(trn_series) == len(target)
    assert trn_series.name == tst_series.name #series란 pandas의 자료구조 중 하나로 리스트와 딕셔너
    temp = pd.concat([trn_series, target], axis=1) #training categorical feature와 target을 좌우로 (
    # Compute target mean
    averages = temp.groupby(by=trn_series.name)[target.name].agg(["mean", "count"]) #category featur
    # Compute smoothing (스무딩은 데이터의 오차를 없애고 원래의 데이터를 추정해내기 위한 기법. 데이터
    smoothing = 1 / (1 + np.exp(-(averages["count"] - min_samples_leaf) / smoothing))
    # Apply average function to all target data
    prior = target.mean() #target의 평균값
    # The bigger the count the less full_avg is taken into account
    averages[target.name] = prior * (1 - smoothing) + averages["mean"] * smoothing
    averages.drop(["mean", "count"], axis=1, inplace=True)
    # Apply averages to trn and tst series
    ft_trn_series = pd.merge(
        trn_series.to_frame(trn_series.name),
        averages.reset_index().rename(columns={'index': target.name, target.name: 'average'}),
        on=trn_series.name,
        how='left')['average'].rename(trn_series.name + '_mean').fillna(prior)
    # pd.merge does not keep the index so restore it
    ft_trn_series.index = trn_series.index
    ft_tst_series = pd.merge(
        tst_series.to_frame(tst_series.name),
        averages.reset_index().rename(columns={'index': target.name, target.name: 'average'}),
        on=tst_series.name,
        how='left')['average'].rename(trn_series.name + '_mean').fillna(prior)
    # pd.merge does not keep the index so restore it
    ft_tst_series.index = tst_series.index
    return add_noise(ft_trn_series, noise_level), add_noise(ft_tst_series, noise_level)
```

Replace ps_car_11_cat with encoded value

Target_encode function을 이용해서 ps_car_11_cat을 train, test dataset에 대해 모두 encoding 된 값으로 대체한다.

In [59]:

```
train_encoded, test_encoded = target_encode(trainset["ps_car_11_cat"],
                              testset["ps_car_11_cat"],
                              target=trainset.target,
                              min_samples_leaf=100,
                              smoothing=10,
                              noise_level=0.01)

trainset['ps_car_11_cat_te'] = train_encoded
trainset.drop('ps_car_11_cat', axis=1, inplace=True)
metadata.loc['ps_car_11_cat','keep'] = False  # Updating the metadata
testset['ps_car_11_cat_te'] = test_encoded
testset.drop('ps_car_11_cat', axis=1, inplace=True)
```

Balance target variable

target variable이 매우 불균형 되어있는 상태이니까 앞선 분석에서 결정한 것처럼 우리는 undersampling을 진행할 것이다.

큰 training set을 가지고 있으니까 undersampling을 진행해보자. 비율은 0.9:0.1로 지정. 이를 통해 우리는 매우 불균형한 결과값 데이터는 10% 미만임을 확인 가능하며, 10% 정도로 undersampling을 진행하게 된다

In [60]:

```
desired_apriori=0.10

# target value의 index 추출
idx_0 = trainset[trainset.target == 0].index
idx_1 = trainset[trainset.target == 1].index

# target value의 기존 record 수 추출
nb_0 = len(trainset.loc[idx_0])
nb_1 = len(trainset.loc[idx_1])

# Undersampling 비율을 계산하고 target==0인 record 수를 계산
undersampling_rate = ((1-desired_apriori)*nb_1)/(nb_0*desired_apriori)
##((1-0.1)*573518) / (0.1*21694)
undersampled_nb_0 = int(undersampling_rate*nb_0)
print('Rate to undersample records with target=0: {}'.format(undersampling_rate))
print('Number of records with target=0 after undersampling: {}'.format(undersampled_nb_0))

# shuffle을 활용하여 undersampling된 개수만큼의 samples를 가지는 nb=0을 무작위 추출
undersampled_idx = shuffle(idx_0, random_state=314, n_samples=undersampled_nb_0)

# 추출한 인덱스와 기존의 idx_1을 활용하여 리스트를 만듦
idx_list = list(undersampled_idx) + list(idx_1)

# Undersample된 데이터 프레임을 돌려받는다
trainset = trainset.loc[idx_list].reset_index(drop=True)
```

```
Rate to undersample records with target=0: 0.34043569687437886
Number of records with target=0 after undersampling: 195246
```

-1을 NaN값으로 대체

이번 데이터셋에서는 결측치값들이 NaN 대신 -1로 되어있기 때문에 이를 NaN으로 바꿔주는 작업이 필요하다.

In [61]:

```
trainset = trainset.replace(-1, np.nan)
testset = testset.replace(-1, np.nan)
```

cat value 더미화

categorical feature에 대해 dummy variable을 만들어줄 예정이다.

더미화란?

출처 : https://kkokkilkon.tistory.com/37 (https://kkokkilkon.tistory.com/37)

1. 더미변수란?
   더미 변수는 범주형 변수를 연속형 변수로 변환한 것인데, 정확히 따지면 연속형변수"스럽게" 만든 것이다.

2. 더미변수는 왜 만드나?
   범주형 변수로는 사용할 수 없고 연속형 변수로만 가능한 분석기법을 사용할 수 있게 해준다.

   예를들어 선형 회귀분석, 로지스틱 회귀분석 등 회귀분석 계열은 원래 설명변수가 연속형 변수여야지 사용할 수 있는 분석기법이다. 하지만 만약 설명변수 중에 범주형 변수가 섞여 있다면, 그 변수를 더미변수로 변환 즉, 연속형 변수스럽게 만들어서 회귀분석을 사용할 수 있다.

3. 더미변수의 특징

   - 더미변수는 0 또는 1의 값을 가진다. 해당 더미변수에 속하면 1 아니면 0의 값을 가진다.
   - 더미변수는 원래 범주형 변수의 범주 개수보다 1개 적게 만들어진다. 예를들어 원래 변수가 성별(남, 여)이라면 남성여부 또는 여성여부 둘 중에 하나만 만든다. (범주의 개수 2개, 더미변수 1개)

   더미변수로 만들어지지 않고 생략되는 범주는 기준이 되는 값이라고 이해하면 된다.

In [62]:

```
cat_features = [a for a in trainset.columns if a.endswith('cat')]

for column in cat_features:
    temp = pd.get_dummies(pd.Series(trainset[column])) # 더미화
    trainset = pd.concat([trainset,temp],axis=1) # axis=1 : 왼쪽+오른쪽으로 가로로 합치기
    trainset = trainset.drop([column],axis=1) # trainset에 대해서 삭제

for column in cat_features:
    temp = pd.get_dummies(pd.Series(testset[column]))
    testset = pd.concat([testset,temp],axis=1)
    testset = testset.drop([column],axis=1) # testset에 대해서 삭제
```

```
In [9]: df_1
Out[9]:
   A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2


In [10]: df_3
Out[10]:
   E   F   G   H
0  A6  B6  C6  D6
1  A7  B7  C7  D7
2  A8  B8  C8  D8
```

```
# concatenating DataFrames along columns, axis=1
In [11]: df_13_axis1 = pd.concat([df_1, df_3], axis=1) # column bind

In [12]: df_13_axis1
Out[12]:
    A   B   C   D   E   F   G   H
0  A0  B0  C0  D0  A6  B6  C6  D6
1  A1  B1  C1  D1  A7  B7  C7  D7
2  A2  B2  C2  D2  A8  B8  C8  D8
```

Drop unused and target columns

trainset에서 target, id삭제, testset에서 id 삭제

In [63]:

```
id_test = testset['id'].values
target_train = trainset['target'].values

#axis=1인 경우 column 삭제, axis=0인 경우 row 삭제
trainset = trainset.drop(['target','id'], axis = 1) # trainset에 대해서 target, id 컬럼 삭제
testset = testset.drop(['id'], axis = 1) # testset에 대해서 id 컬럼 삭제
```

전처리가 완료된 training, test set을 살펴보자

In [65]:

```
print("Train dataset (rows, cols):",trainset.values.shape, "₩nTest dataset (rows, cols):",testset.va
```

```
Train dataset (rows, cols): (216940, 91)
Test dataset (rows, cols): (616098, 91)
```

모델 준비

앙상블 기법 + Cross validation
데이터를 KFold로 분리하기 위해 Ensemble 클래스를 준비. 모델을 훈련 시킨 후 결과를 앙상블 한다.

Class의 init method에서는 총 4개의 파라미터를 받는다

- self : 초기화될 object
- n_splits : cross validation에서 나뉘개 될 split 수
- stacker : base_model들의 예측결과를 stacking하기 위한 것
- base_model : training에 사용되는 base model

fit_predict에서는 4가지 기능을 지원한다

- training data를 n_split개의 fold로 나눈다
- 각 fold에 대해 base model들을 돌린다
- 각 모델들에 대한 예측을 수행한다
- stacker를 이용해 각 모델의 결과를 쌓는다

K Fold의 효과

- 어떤 데이터세트가 과대적합을 일으켰는지 확인 가능 (과대적합을 막을 수 있음)
- KFold를 쓰면 어느 지점에서 학습이 덜 되는지 더 되는지 아 ㄹ수 있으니까 파라미터 값을 조절할 수 있다고 함

Stratified cross validation

데이터가 편향되어 있을 경우 단순 KFold 를 사용하면 성능 평가가 잘 되지 않을 수 있다. 이럴 때 stratified k fold cross validation을 사용한다고 함.

- n_splits : 몇 개로 분할할지
- shuffle : True를 넣으면 Fold를 나누기 전에 무작위로 섞음

In [66]:

```
class Ensemble(object):
    def __init__(self, n_splits, stacker, base_models):
        self.n_splits = n_splits
        self.stacker = stacker
        self.base_models = base_models

#X : trainset
#y : target_train
#T : testset
    def fit_predict(self, X, y, T):
        X = np.array(X)
        y = np.array(y)
        T = np.array(T)

        folds = list(StratifiedKFold(n_splits=self.n_splits, shuffle=True, random_state=314).split(

        S_train = np.zeros((X.shape[0], len(self.base_models))) #row : Train dataset row, #col : 사
        S_test = np.zeros((T.shape[0], len(self.base_models)))
        for i, clf in enumerate(self.base_models):

            S_test_i = np.zeros((T.shape[0], self.n_splits))

            for j, (train_idx, test_idx) in enumerate(folds):
                X_train = X[train_idx]
                y_train = y[train_idx]
                X_holdout = X[test_idx]


                print ("Base model %d: fit %s model | fold %d" % (i+1, str(clf).split('(')[0], j+1)
                clf.fit(X_train, y_train)
                cross_score = cross_val_score(clf, X_train, y_train, cv=3, scoring='roc_auc')
                print("cross_score [roc-auc]: %.5f [gini]: %.5f" % (cross_score.mean(), 2*cross_scor
                y_pred = clf.predict_proba(X_holdout)[:,1]

                S_train[test_idx, i] = y_pred
                S_test_i[:, j] = clf.predict_proba(T)[:,1]
            S_test[:, i] = S_test_i.mean(axis=1)

        results = cross_val_score(self.stacker, S_train, y, cv=3, scoring='roc_auc')
        # Calculate gini factor as 2 * AUC - 1
        print("Stacker score [gini]: %.5f" % (2 * results.mean() - 1))

        self.stacker.fit(S_train, y)
        res = self.stacker.predict_proba(S_test)[:,1]
        return res
```

Base model을 위한 파라미터

우리는 3가지의 LightGBM 모델과 하나의 XGBoost model을 사용할 것이다.
각 모델들은 데이터 훈련에 사용된다.(cross-validation의 fold 값은 3)

In [67]:

```
# LightGBM params
# lgb_1
lgb_params1 = {}
lgb_params1['learning_rate'] = 0.02 # 부스팅 스텝을 반복적으로 수행할 때 업데이트 되는 학습률. (0~1
lgb_params1['n_estimators'] = 650 # 만드는 tree의 개수 개수
lgb_params1['max_bin'] = 10 # 히스토그램 빈 개수. 규제(regularization)로도 동작하게 되므로 모델의 여
lgb_params1['subsample'] = 0.8 # 발췌하는 데이터 비율
lgb_params1['subsample_freq'] = 10 # iteration 몇번째에 해당 데이터를 업데이트 할지
lgb_params1['colsample_bytree'] = 0.8    # 컬럼에 대한 샘플링을 통해 각각의 다양성을 높임. 랜덤 포러
lgb_params1['min_child_samples'] = 500 # 최종 결정 클래스인 leaf node가 되기 위해서 최소한으로 필요
lgb_params1['seed'] = 314
lgb_params1['num_threads'] = 4 # 병렬처리 시 처리할 쓰레드 기본값. 사용 가능한 코어 수 그대로 적거니

# lgb2
lgb_params2 = {}
lgb_params2['n_estimators'] = 1090
lgb_params2['learning_rate'] = 0.02
lgb_params2['colsample_bytree'] = 0.3
lgb_params2['subsample'] = 0.7
lgb_params2['subsample_freq'] = 2
lgb_params2['num_leaves'] = 16
lgb_params2['seed'] = 314
lgb_params2['num_threads'] = 4

# lgb3
lgb_params3 = {}
lgb_params3['n_estimators'] = 1100
lgb_params3['max_depth'] = 4
lgb_params3['learning_rate'] = 0.02
lgb_params3['seed'] = 314
lgb_params3['num_threads'] = 4

# XGBoost params
xgb_params = {}
xgb_params['objective'] = 'binary:logistic'
xgb_params['learning_rate'] = 0.04
xgb_params['n_estimators'] = 490
xgb_params['max_depth'] = 4
xgb_params['subsample'] = 0.9
xgb_params['colsample_bytree'] = 0.9
xgb_params['min_child_weight'] = 10
xgb_params['num_threads'] = 4
```

파라미터를 이용해 모델들을 초기화한다

우리는 3개의 base model을 이용하고 그걸 쌓아 올린다. base model의 경우 위에서 정의해준 파라미터를 따르게 된다.

In [68]:

```
# Base models
lgb_model1 = LGBMClassifier(**lgb_params1)

lgb_model2 = LGBMClassifier(**lgb_params2)

lgb_model3 = LGBMClassifier(**lgb_params3)

xgb_model = XGBClassifier(**xgb_params)

# Stacking model
log_model = LogisticRegression() #Binary classification 을 해야하니까 LogisticRegression 사용
```

앙상블 object 초기화

In [69]:

```
stack = Ensemble(n_splits=3,
        stacker = log_model,
        base_models = (lgb_model1, lgb_model2, lgb_model3, xgb_model))
```

예측 모델 돌리기

stack object의 fit_predict method를 이용함으로써 base model들을 training에 사용하게 된다. 각 모델에서 target 값을 예측하게 되고 각 결과를 stacker를 이용해서 ensemble하게 된다.

In [70]:

```
y_prediction = stack.fit_predict(trainset, target_train, testset)
```

```
Base model 1: fit LGBMClassifier model | fold 1
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
cross_score [roc-auc]: 0.63729 [gini]: 0.27457
Base model 1: fit LGBMClassifier model | fold 2
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
cross_score [roc-auc]: 0.63971 [gini]: 0.27943
Base model 1: fit LGBMClassifier model | fold 3
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
ue: num_threads=4
[LightGBM] [Warning] num_threads is set=4, n_jobs=-1 will be ignored. Current val
```

스태킹 앙상블(stacking ensemble)이론

출처 : https://lsjsj92.tistory.com/559?category=853217 (https://lsjsj92.tistory.com/559?category=853217)
CV 기반의 스태킹은 각 모델들이 교차 검증(KFold 등)으로 최종 모델을 위한 학습용 데이터를 생성합니다. 또한, 예측을 위한 테스트용 데이터도 생성하여 이를 기반으로 최종 모델이 학습을 진행하게 됩니다.

1. 데이터를 Fold로 나눔
2. 각 모델 별로 Fold로 나누어진 데이터를 기반으로 훈련을 진행 (1) 이때 각 Fold마다 뽑아진 훈련 데이터로 모델을 훈련하고 검증 데이터를 활용해 예측 후 값을 저장 (2) 마찬가지로 각 Fold 마다 나온 model을 기반으로 원본 X_test 데이터를 훈련하여 저장(이것은 추후 Average 됨)
3. 2까지 진행해서 나온 각 모델별 예측 데이터(2-1)를 모두 stacking 하여 최종 모델의 훈련 데이터로 사용 (1) label은 원본 y_train값으로 진행
4. 2-2 에서 나온 데이터로 예측을 수행하여 pred값을 뽑아냄
5. 4에서 나온 pred와 y_test값을 비교해서 최종 모델 평가

In [ ]: