Machine Learning Engineer Nanodegree

Capstone Project

Rafael Paulino
February 10th, 2018

Porto Seguro's Safe Driver Prediction

I. Definition

Project Overview

"Nothing ruins the thrill of buying a brand new car more quickly than seeing your new insurance bill. The sting's even more painful when you know you're a good driver. It doesn't seem fair that you have to pay so much if you've been cautious on the road for years.

Porto Seguro, one of Brazil's largest auto and homeowner insurance companies, completely agrees. Inaccuracies in car insurance company's claim predictions raise the cost of insurance for good drivers and reduce the price for bad ones."

We cannot say that the the use of machine learning to solve this kind of problem (insurance claim prediction problem) is a new one. For example, Chapados et al (2001) has already studied the problem but, instead of trying to predict if one will or not claim the insurance, it tried to estimate insurance premia (a regression problem). There is also news articles, for instance "How AI And Machine Learning Are Used To Transform The Insurance Industry" published on Forbes on 24th October 2017, showing how the industry is changing with the use of Machine Learning.

In this project we will be using machine learning techniques to try predicting the probability that a driver will initiate an auto insurance claim in the next year.

Actually, this problem was proposed as a competition on Kaggle, where we have access to historical information about drivers and related insurance claim.

Problem Statement

In Machine Learning, there are two big kinds of learning: *supervised* and *unsupervised*. Among the first group, there are *regression* and *classification* problems. In this project, we will create a **supervised** learning model that is able to **classify** based on data if a driver is more likely to initiate an auto insurance claim.

To build a classification supervised learning model we need a dataset that contains historical data about the domain and its classification. In our case, we need information like car details (brand, model, age, type, etc), driver details (age, living place, workplace, if is it used to work, gender, marital status, etc.), historical or related information (has this driver already claimed auto insurance before? How safe is the neighborhood/city where the driver lives? How far is the his workplace from home? etc.) and, for each one if, in that year, he had started or not an auto insurance claim (we call that "label" or "target").

In the dataset provided by Porto Seguro in the kaggle competition, we don't know what exactly mean each piece of information (for security and privacy reasons), but that will not be a problem. Machine Learning algorithms find patterns among the data. In our case, it will find the relation between the features (individual, car, registration and calculated data) and the target (claimed or not).

Among other algorithms suited for the task, I selected **XGBoost** to create a classification model. This algorithm is an implementation of gradient boosted decision trees designed for speed and performance. It has been proven to be a nice choice to be used with structured data (that's our case!) in many kaggle competitions. This is also a good choice for our problem because our dataset is very unbalanced (we will discuss more about that in the next section), what means: it is much more likely that an auto insurance is not claimed.

To be able to compare the results of our project we will be using ROC AUC (explained below). For example, if we always predict that there will be no claim, we will get a score of 0.5. If we could always predict the future, we would get a score of 1.0. Using a naive bayes learner (a very simple algorithm), I got 0.58. We expect to get a higher score than that in this project.

Metrics

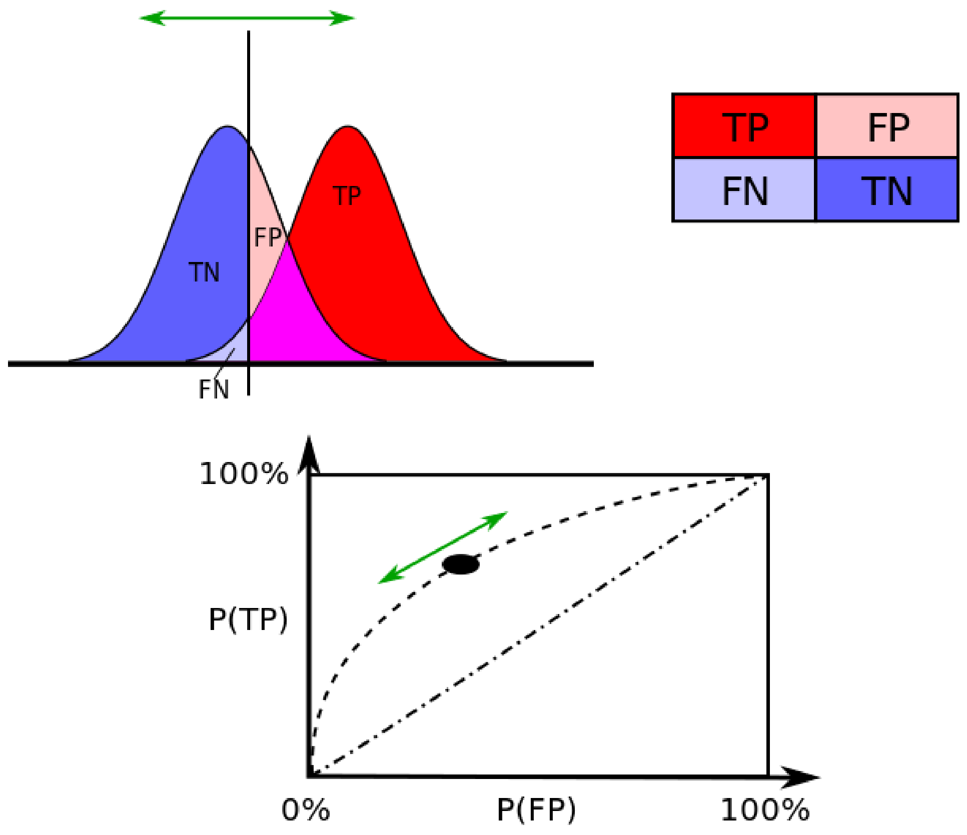
As I said before, we will be using **ROC AUC** or Area Under the Receiver Operating Characteristic Curve. For our scenario, where the labels (or targets) are skewed, this is a better scorer than accuracy. Accuracy would be a poor choice because if I create a model that always predicts that someone would NOT claim the insurance, it would get a performance around 95% (which is a great performance by the way). But, at the same time, that tell us nothing (as we know that around 95% of the time insurance is not claimed). So we need to care about true and false positives. ROC AUC is insensitive to unbalanced data as it uses the relation between true and false positives.

A ROC space is defined by FPR and TPR as *x* and *y* axes, respectively:

$$FPR = \frac{FP}{FP + TN} \qquad TPR = \frac{TP}{TP + FN}$$

Where:

FP: False Positive	TP: True Positive
FN: False Negative	TN: True Negative
FPR: False Positive Rate	TPR: True Positive Rate



By Sharpr - Own work, CC BY-SA 3.0, [*https://commons.wikimedia.org/w/index.php?curid=44059691*](https://commons.wikimedia.org/w/index.php?curid=44059691)

The best possible prediction (perfect classification) method would yield a point in the upper left corner or coordinate (0, 1) of the ROC space (no False Negatives and no False Positives); along the diagonal axis (a.k.a. line of no-discrimination) lies random guess models. That way, good classifiers would be above the diagonal line and bad classifiers below.

Given *X*, the estimated probability of a class, and a threshold *T*, the instance is classified as "positive" if *X* > *T* and "negative" otherwise. *X* follows a probability density $f_1(x)$ if the instance actually belongs to class "positive" and $f_0(x)$ if otherwise. Therefore:

$$TPR(T) = \int_{T}^{\infty} f_1(x)dx \qquad FPR(T) = \int_{T}^{\infty} f_0(x)dx$$

That way, we can plot the ROC curve by varying T parameter.

Therefore, the Area Under the Curve (AUC) is given by:

$$A = \int_{\infty}^{-\infty} TPR(T)(-FPR'(T))dT$$

One important relation for our problem is ROC AUC and Gini coefficient (G_1), as the later is the selected metric on the referred kaggle competition:

$$G_1 = 2AUC - 1$$

and

$$G_1 = 1 - \sum_{k=1}^{n} (X_k - X_{k-1})(Y_k + Y_{k-1})$$

Hand, David J.; and Till, Robert J. (2001); A simple generalization of the area under the ROC curve for multiple class classification problems, Machine Learning, 45, 171–186.

You can find more information about ROC AUC on Wikipedia "Receiver operating characteristic", "Compute Area Under the Receiver Operating Characteristic Curve" and "A very simple explanation for AUC or Area Under Curve (ROC Curve)".

II. Analysis

Data Exploration

The dataset that will be use can be found at a Kaggle Competition named "Porto Seguro's Safe Driver Prediction". For privacy reasons, Porto Seguro did not shared the meaning of each input feature, but it seems that they are related to the individual, registration, car or calculated information. The "target" feature is given and it is as simple as 1 when that customer filled claim request and 0 otherwise.

I'm not supplying sample data because there are a lot of columns. Therefore, I'll try to give you some intuition about the data and its values.

1. It consists of **58 input fields** and **1 target** over **595.212 samples**.
2. From the input fields we can say that: (take a look at the following table)
    a. 1 is the "id", an unique identifier of an insurance application
    b. 18 are about the individual;
    c. 3 are about the registration;
    d. 16 are about the car;
    e. 20 are calculated;
    f. 14 are categorical data;
    g. 17 are binary (0 or 1);
    h. 26 are numbers (integer or real);

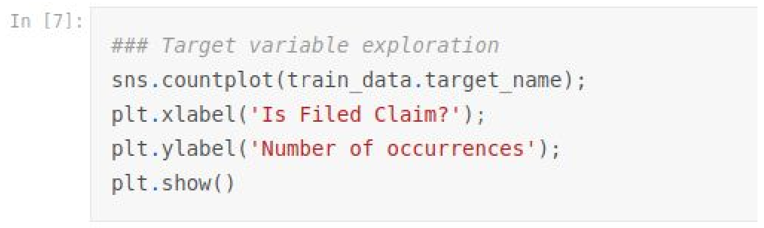
	individual	**registration**	**car**	**calculated**
categorical	ps_ind_02_cat ps_ind_04_cat ps_ind_05_cat	-	ps_car_01_cat ps_car_02_cat ps_car_03_cat ps_car_04_cat ps_car_05_cat ps_car_06_cat ps_car_07_cat ps_car_08_cat ps_car_09_cat ps_car_10_cat ps_car_11_cat	-
binary	ps_ind_06_bin ps_ind_07_bin ps_ind_08_bin ps_ind_09_bin ps_ind_10_bin ps_ind_11_bin ps_ind_12_bin ps_ind_13_bin ps_ind_16_bin ps_ind_17_bin ps_ind_18_bin	-	-	ps_calc_15_bin ps_calc_16_bin ps_calc_17_bin ps_calc_18_bin ps_calc_19_bin ps_calc_20_bin

numbers	ps_ind_01 ps_ind_03 ps_ind_14 ps_ind_15	ps_reg_01 ps_reg_02 ps_reg_03	ps_car_11 ps_car_12 ps_car_13 ps_car_14 ps_car_15	ps_calc_01 ps_calc_02 ps_calc_03 ps_calc_04 ps_calc_05 ps_calc_06 ps_calc_07 ps_calc_08 ps_calc_09 ps_calc_10 ps_calc_11 ps_calc_12 ps_calc_13 ps_calc_14

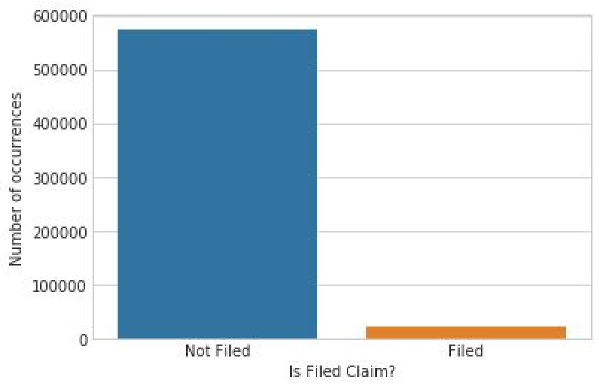
3. There are 13 features with some missing values indicated by -1.

Feature name	**Count of missing records**	**%**
ps_ind_02_cat	216	0.04
ps_ind_04_cat	83	0.01
ps_ind_05_cat	5.809	0.98
ps_reg_03	107.772	18.11
ps_car_01_cat	107	0.02
ps_car_02_cat	5	0.00
ps_car_03_cat	411.231	69.09
ps_car_05_cat	266.551	44.78
ps_car_07_cat	11.489	1.93
ps_car_09_cat	569	0.10
ps_car_11	5	0.00
ps_car_12	1	0.00
ps_car_14	42.620	7.16

4. The dataset is very unbalanced as we can see in the graph below. Only 3.64% (21.694 samples over 595.212) are classified as "1".



```
In [7]:
### Target variable exploration
sns.countplot(train_data.target_name);
plt.xlabel('Is Filed Claim?');
plt.ylabel('Number of occurrences');
plt.show()
```



Source: https://www.kaggle.com/neviadomski/data-exploration-porto-seguro-s-safe-driver (thanks to Sergei Neviadomski)

5. Categorical data

Feature name	Distinct values	Feature name	Distinct Values
ps_car_01_cat	12	ps_car_08_cat	2
ps_car_02_cat	2	ps_car_09_cat	5
ps_car_03_cat	2	ps_car_10_cat	3
ps_car_04_cat	10	ps_car_11_cat	104
ps_car_05_cat	2	ps_ind_02_cat	4
ps_car_06_cat	18	ps_ind_04_cat	2
ps_car_07_cat	2	ps_ind_05_cat	7

6. Numerical data statistics

	ps_ind_01	ps_ind_03	ps_ind_14	ps_ind_15	ps_reg_01
mean	1.900378	4.423318	0.012451	7.299922	0.610991
std	1.983789	2.699902	0.127545	3.546042	0.287643
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	0.000000	5.000000	0.400000
50%	1.000000	4.000000	0.000000	7.000000	0.700000
75%	3.000000	6.000000	0.000000	10.000000	0.900000
max	7.000000	11.000000	4.000000	13.000000	0.900000

	ps_reg_02	ps_reg_03	ps_car_11	ps_car_12	ps_car_13
mean	0.439184	0.894047	2.346105	0.379947	0.813265
std	0.404264	0.312581	0.832493	0.058300	0.224588
min	0.000000	0.061237	0.000000	0.100000	0.250619
25%	0.200000	0.666615	2.000000	0.316228	0.670867
50%	0.300000	0.894047	3.000000	0.374166	0.765811
75%	0.600000	1.000000	3.000000	0.400000	0.906190
max	1.800000	4.037945	3.000000	1.264911	3.720626

	ps_car_14	ps_car_15	ps_calc_01	ps_calc_02	ps_calc_03
mean	0.374691	3.065899	0.449756	0.449589	0.449849
std	0.043947	0.731366	0.287198	0.286893	0.287153
min	0.109545	0.000000	0.000000	0.000000	0.000000
25%	0.353553	2.828427	0.200000	0.200000	0.200000
50%	0.374691	3.316625	0.500000	0.400000	0.500000
75%	0.396485	3.605551	0.700000	0.700000	0.700000
max	0.636396	3.741657	0.900000	0.900000	0.900000

	ps_calc_04	ps_calc_05	ps_calc_06	ps_calc_07	ps_calc_08
mean	2.372081	1.885886	7.689445	3.005823	9.225904
std	1.117219	1.134927	1.334312	1.414564	1.459672
min	0.000000	0.000000	0.000000	0.000000	2.000000
25%	2.000000	1.000000	7.000000	2.000000	8.000000
50%	2.000000	2.000000	8.000000	3.000000	9.000000
75%	3.000000	3.000000	9.000000	4.000000	10.000000
max	5.000000	6.000000	10.000000	9.000000	12.000000

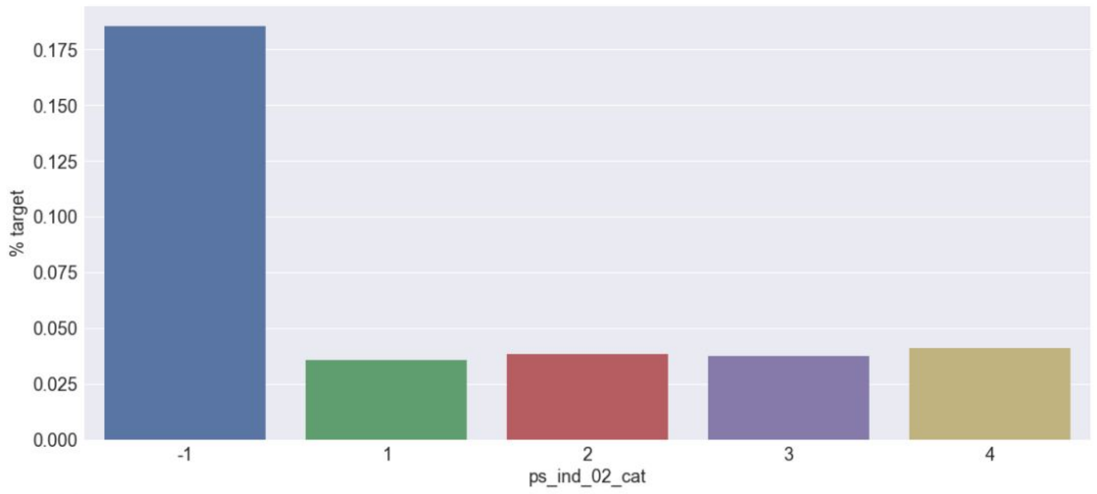
	ps_calc_09	ps_calc_10	ps_calc_11	ps_calc_12	ps_calc_13	ps_calc_14
mean	2.339034	8.433590	5.441382	1.441918	2.872288	7.539026
std	1.246949	2.904597	2.332871	1.202963	1.694887	2.746652
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1.000000	6.000000	4.000000	1.000000	2.000000	6.000000
50%	2.000000	8.000000	5.000000	1.000000	3.000000	7.000000
75%	3.000000	10.000000	7.000000	2.000000	4.000000	9.000000
max	7.000000	25.000000	19.000000	10.000000	13.000000	23.000000

From the information about the data collected above:

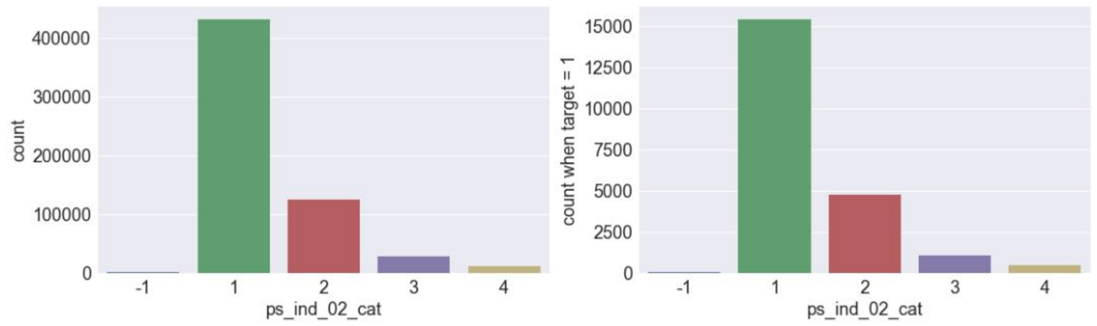
- We don't need to handle missing data, because XGBoost can handle it;
- Might not be necessary to encode categorical data, because XGBoost can handle categorical data as it uses trees as weak learners;
- We also don't need to scale numeric feature for the same reason;

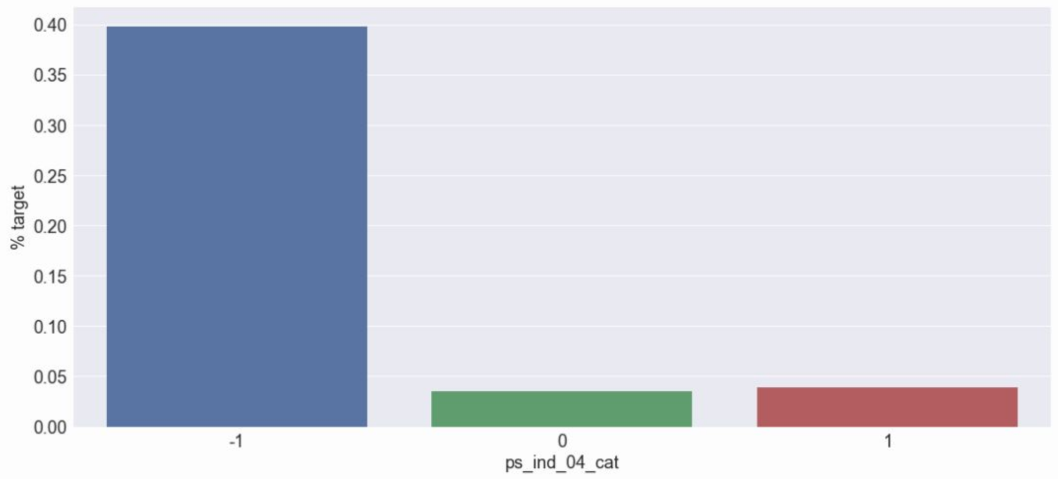
Exploratory Visualization

In the following pages you will see three plots for each categorical feature. The first is percentage of *"target = 1"* for each class. The second graph (second row, left side) is the distribution of the feature in the whole dataset and the third (second row, right side) is the distribution of the feature in the subset when *"target = 1"*.

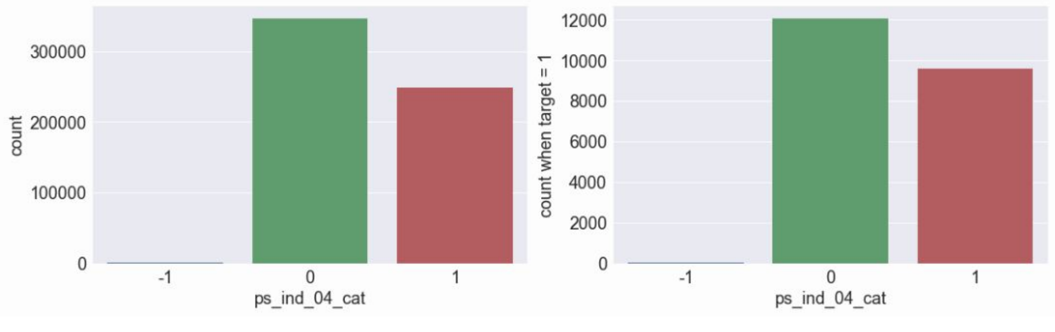

% target
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
-1
1
2
3
4
ps_ind_02_cat

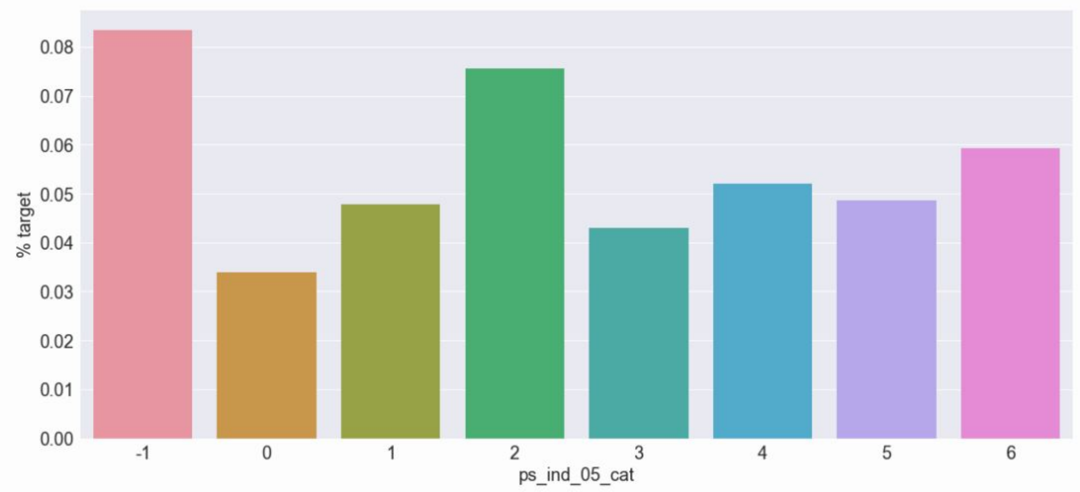
```
<matplotlib.figure.Figure at 0x7f5351a564e0>
```


count
0
100000
200000
300000
400000
-1
1
2
3
4
ps_ind_02_cat
count when target = 1
0
2500
5000
7500
10000
12500
15000
-1
1
2
3
4
ps_ind_02_cat

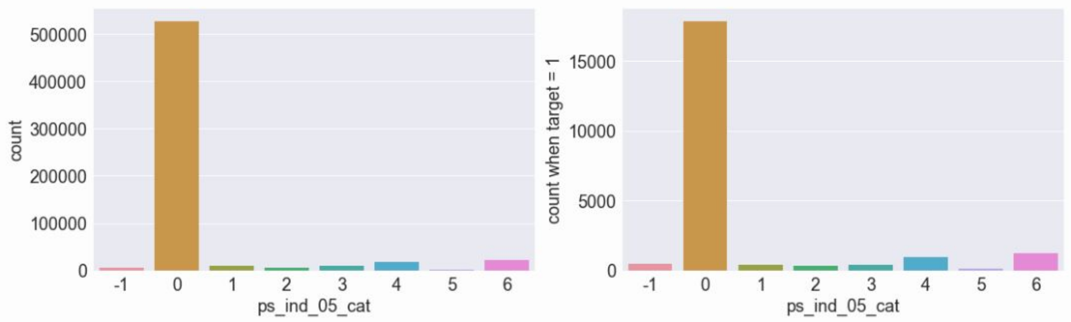

% target
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
-1
0
1
ps_ind_04_cat

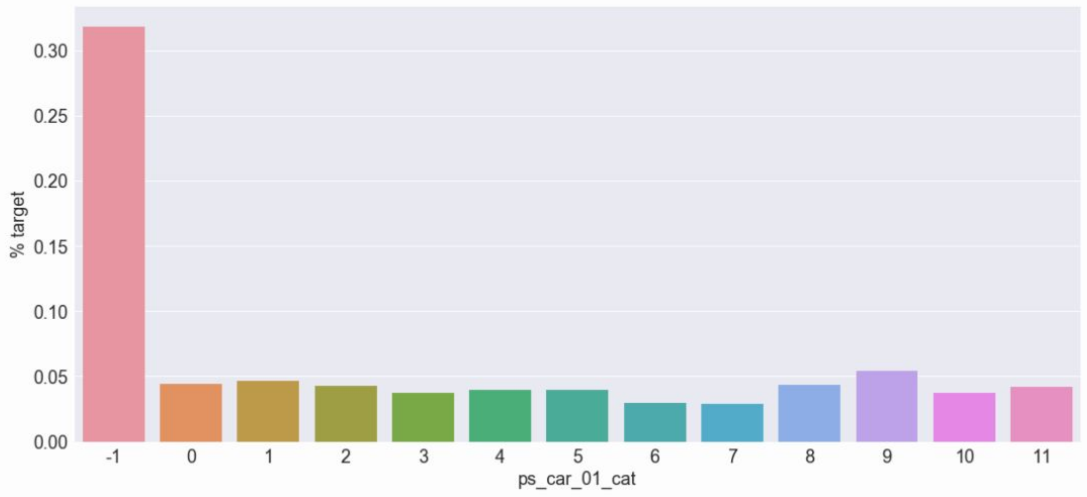
```
<matplotlib.figure.Figure at 0x7f5351d95278>
```


count
0
100000
200000
300000
-1
0
1
ps_ind_04_cat
count when target = 1
0
2000
4000
6000
8000
10000
12000
-1
0
1
ps_ind_04_cat

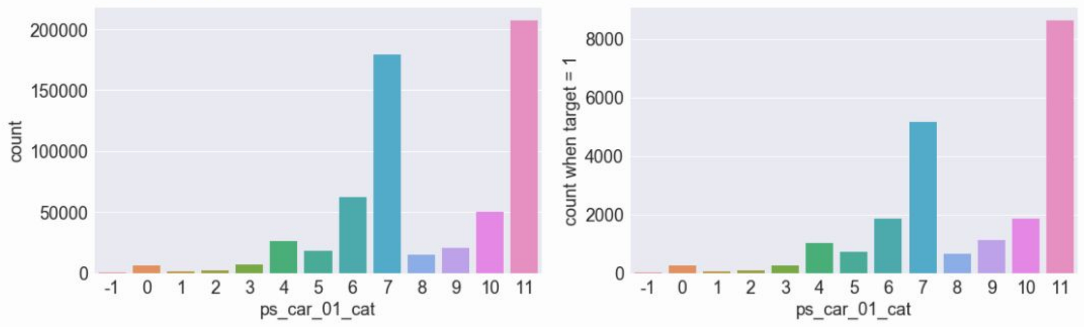

% target
ps_ind_05_cat
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
-1
0
1
2
3
4
5
6

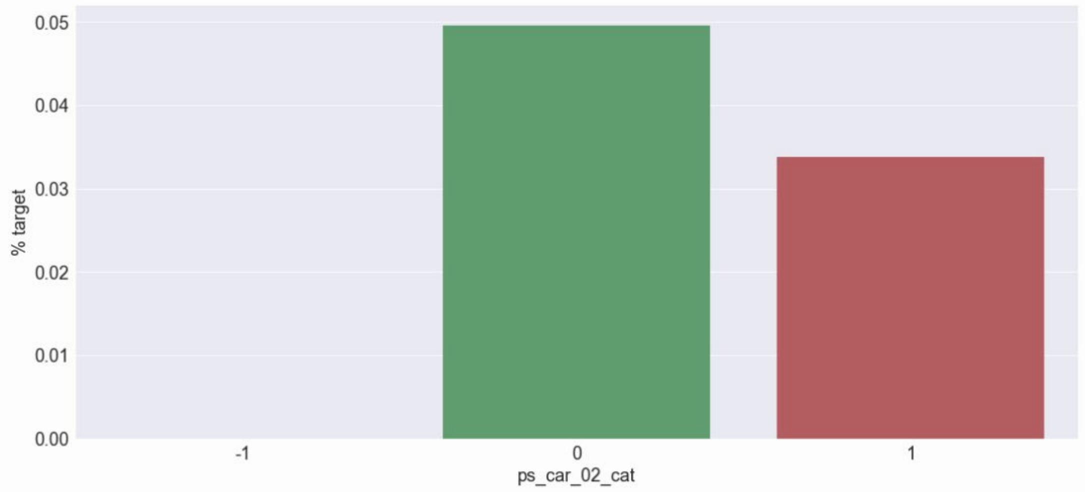
```
<matplotlib.figure.Figure at 0x7f5351e68ac8>
```


count
count when target = 1
ps_ind_05_cat
ps_ind_05_cat
0
100000
200000
300000
400000
500000
5000
10000
15000
-1
0
1
2
3
4
5
6

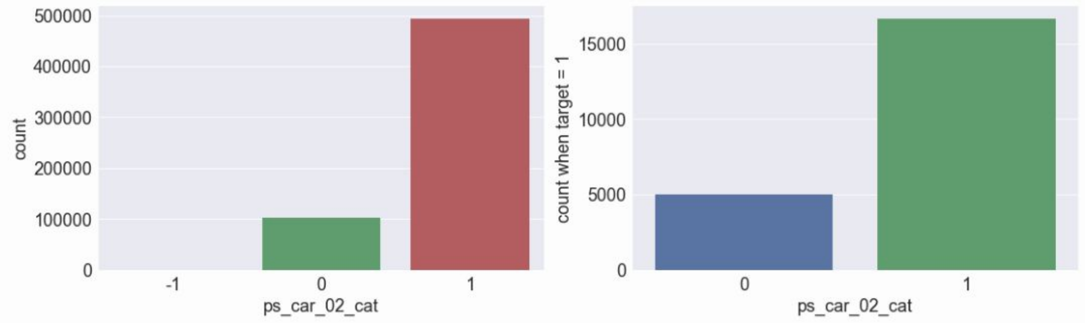

% target
ps_car_01_cat
0.00
0.05
0.10
0.15
0.20
0.25
0.30
-1
0
1
2
3
4
5
6
7
8
9
10
11

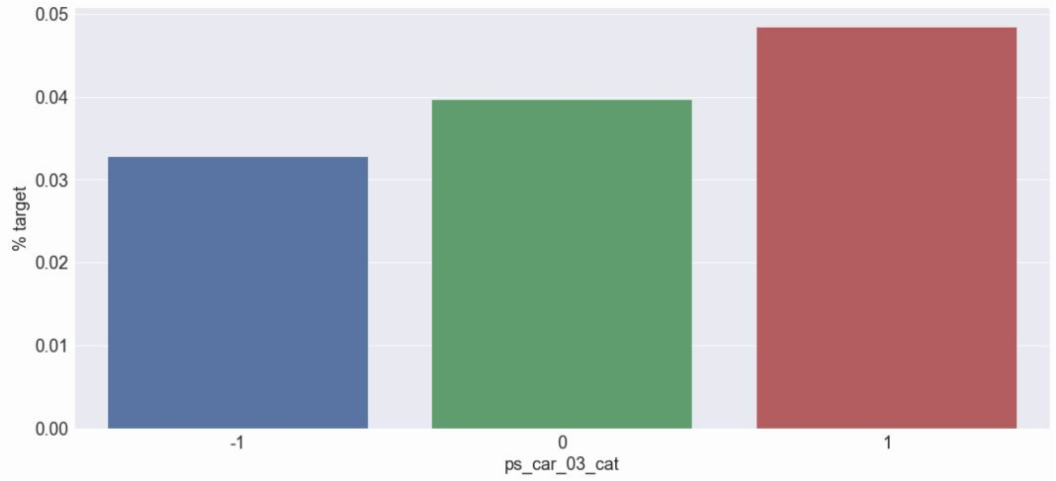
```
<matplotlib.figure.Figure at 0x7f53518487b8>
```


count
count when target = 1
ps_car_01_cat
ps_car_01_cat
0
50000
100000
150000
200000
2000
4000
6000
8000
-1
0
1
2
3
4
5
6
7
8
9
10
11

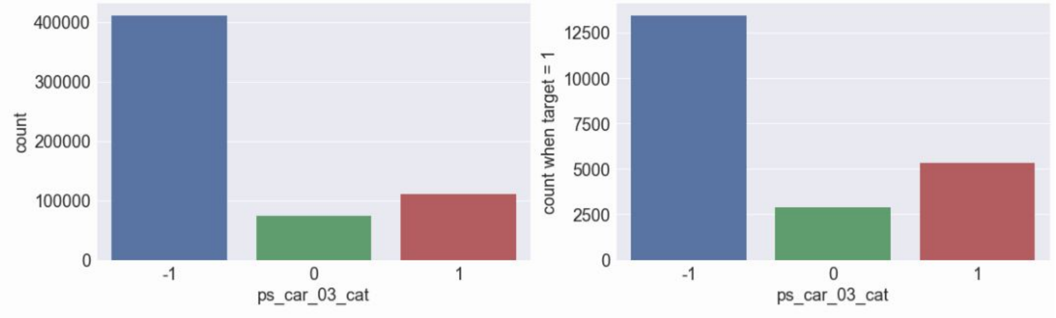

0.05
0.04
0.03
0.02
0.01
0.00
% target
-1
0
1
ps_car_02_cat

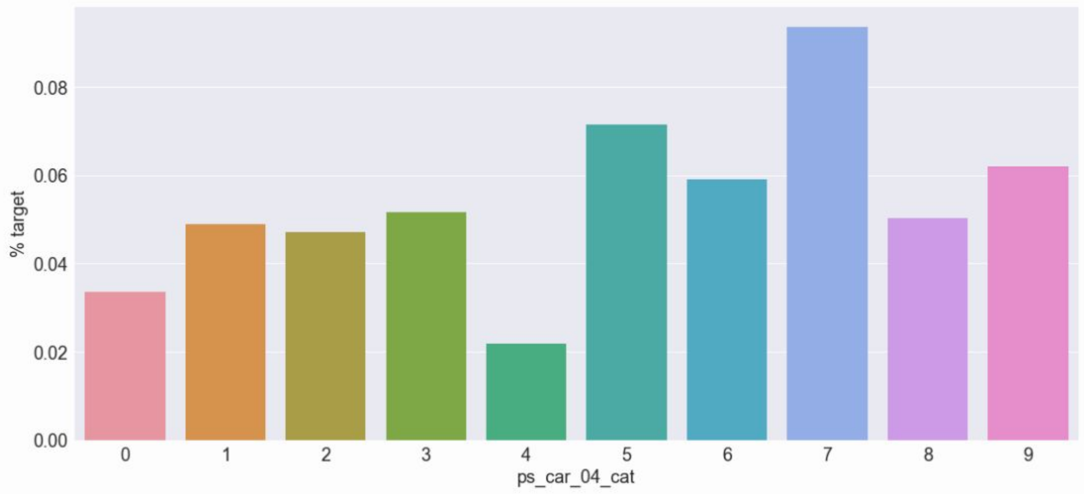
```
<matplotlib.figure.Figure at 0x7f5351531e48>
```


500000
400000
300000
200000
100000
0
count
-1
0
1
ps_car_02_cat
15000
10000
5000
0
count when target = 1
0
1
ps_car_02_cat

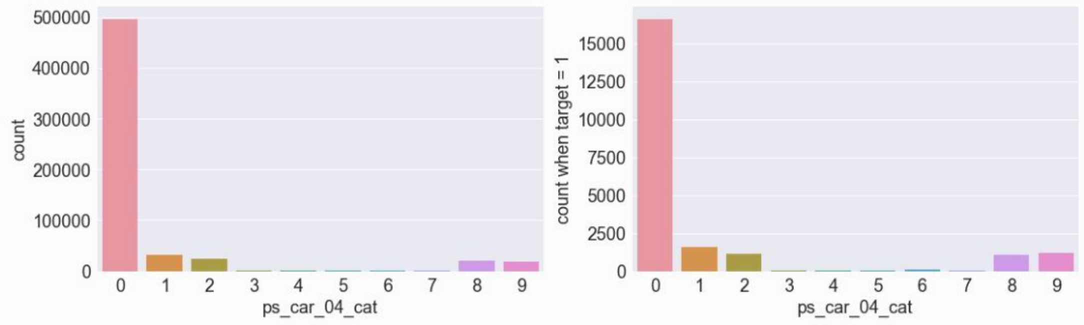

0.05
0.04
0.03
0.02
0.01
0.00
% target
-1
0
1
ps_car_03_cat

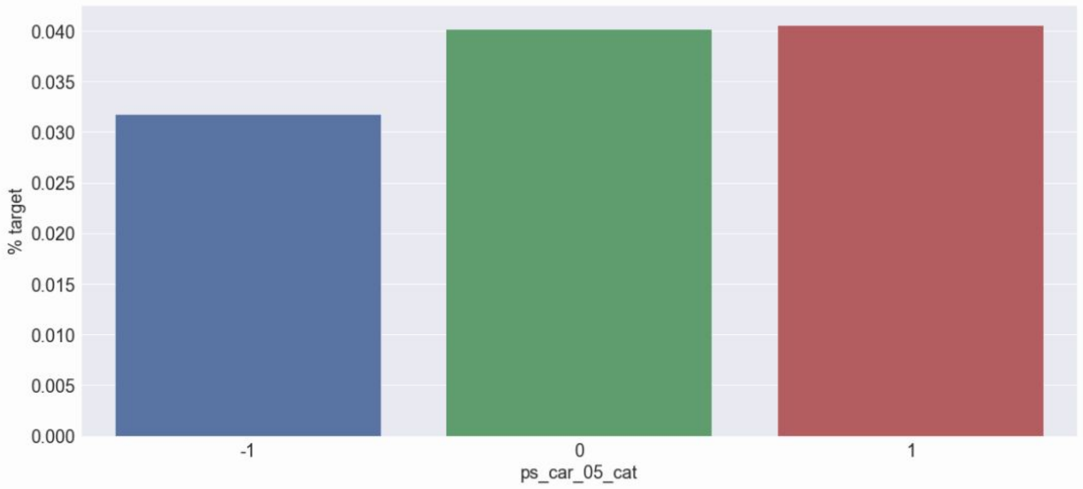
```
<matplotlib.figure.Figure at 0x7f535175eef0>
```


400000
300000
200000
100000
0
count
-1
0
1
ps_car_03_cat
12500
10000
7500
5000
2500
0
count when target = 1
-1
0
1
ps_car_03_cat

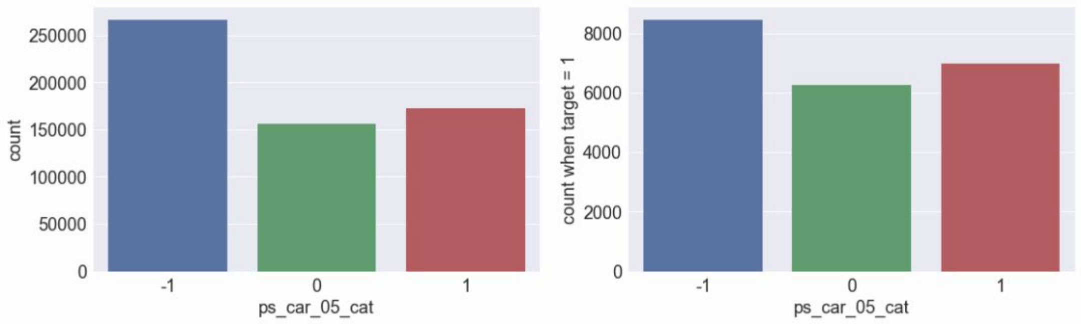

% target
0.00
0.02
0.04
0.06
0.08
0
1
2
3
4
5
6
7
8
9
ps_car_04_cat

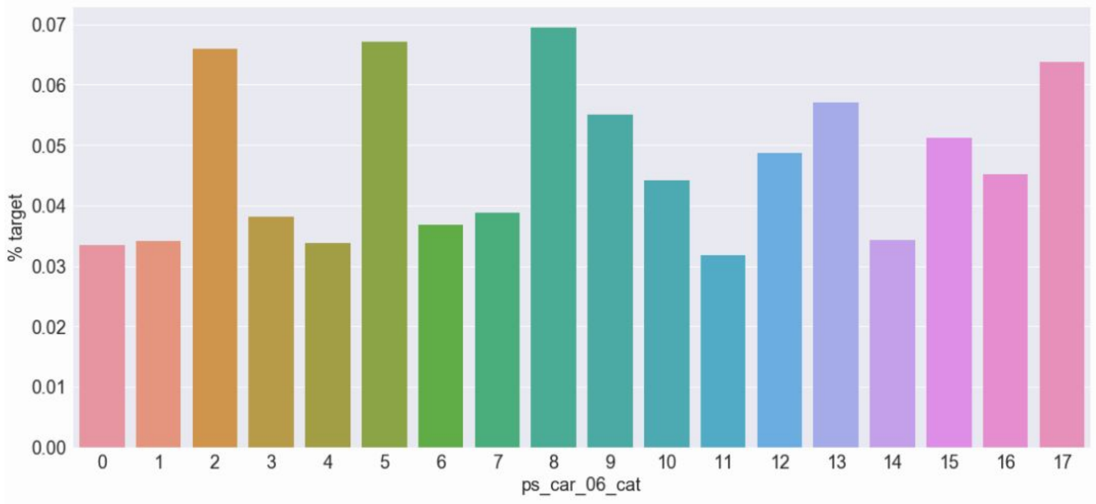
```
<matplotlib.figure.Figure at 0x7f535175e6a0>
```


count
0
100000
200000
300000
400000
500000
0
1
2
3
4
5
6
7
8
9
ps_car_04_cat
count when target = 1
0
2500
5000
7500
10000
12500
15000
ps_car_04_cat

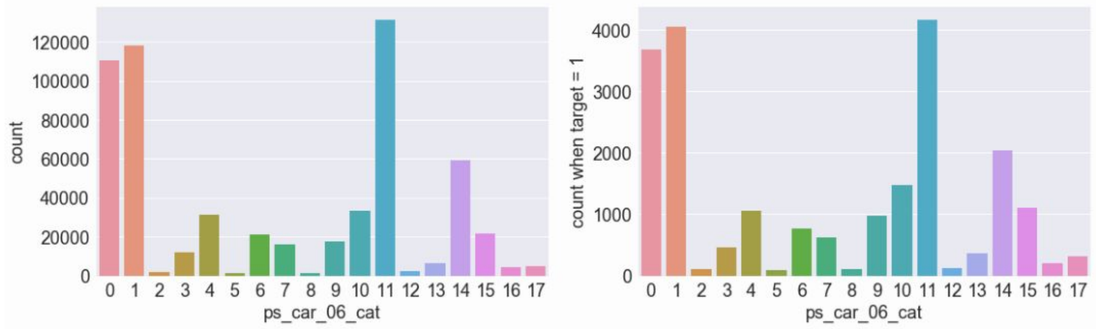

% target
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0.040
-1
0
1
ps_car_05_cat

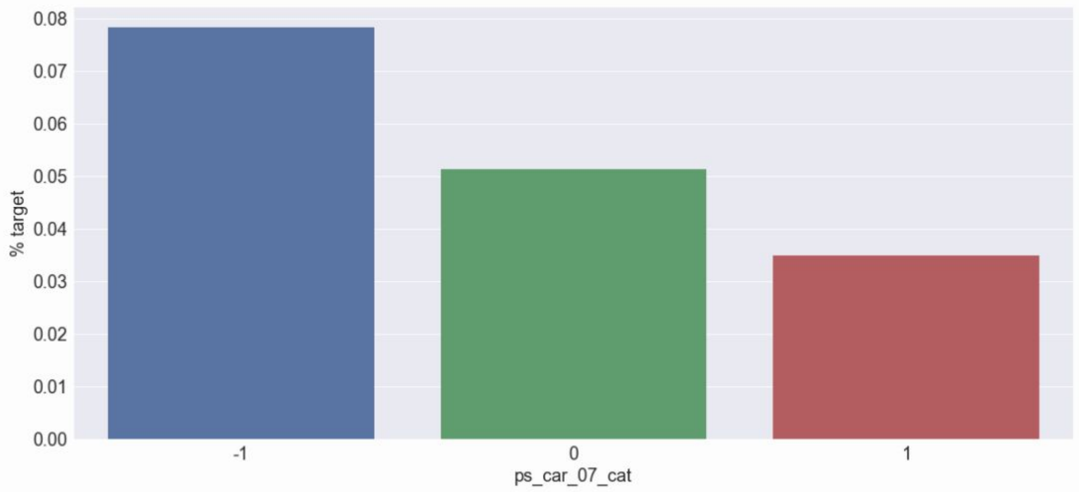
```
<matplotlib.figure.Figure at 0x7f5351a452e8>
```


count
0
50000
100000
150000
200000
250000
-1
0
1
ps_car_05_cat
count when target = 1
0
2000
4000
6000
8000
-1
0
1
ps_car_05_cat

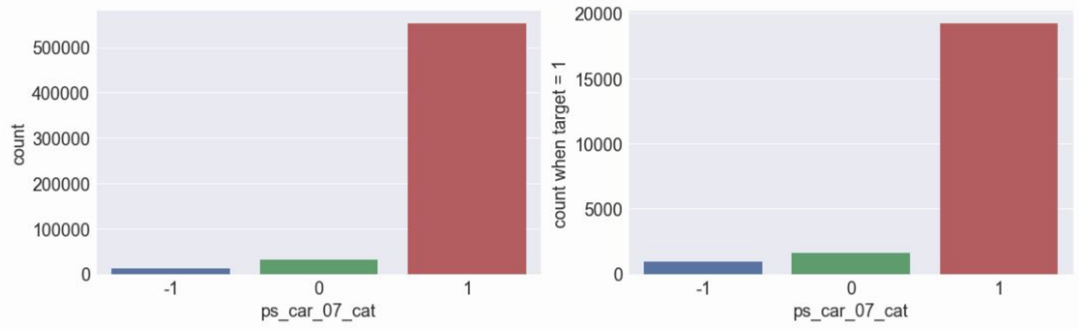

% target
ps_car_06_cat

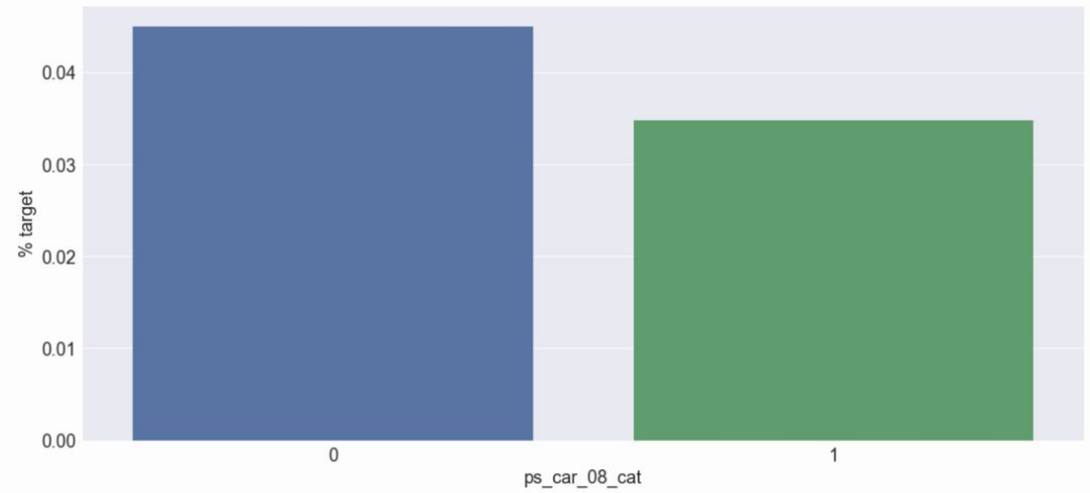
```
<matplotlib.figure.Figure at 0x7f5351856cc0>
```


count
count when target = 1
ps_car_06_cat
ps_car_06_cat


% target
ps_car_07_cat

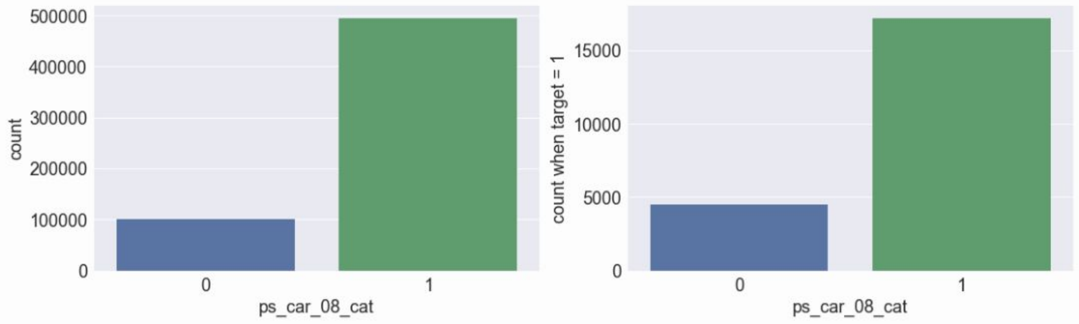
```
<matplotlib.figure.Figure at 0x7f5351c9f198>
```


count
count when target = 1
ps_car_07_cat
ps_car_07_cat

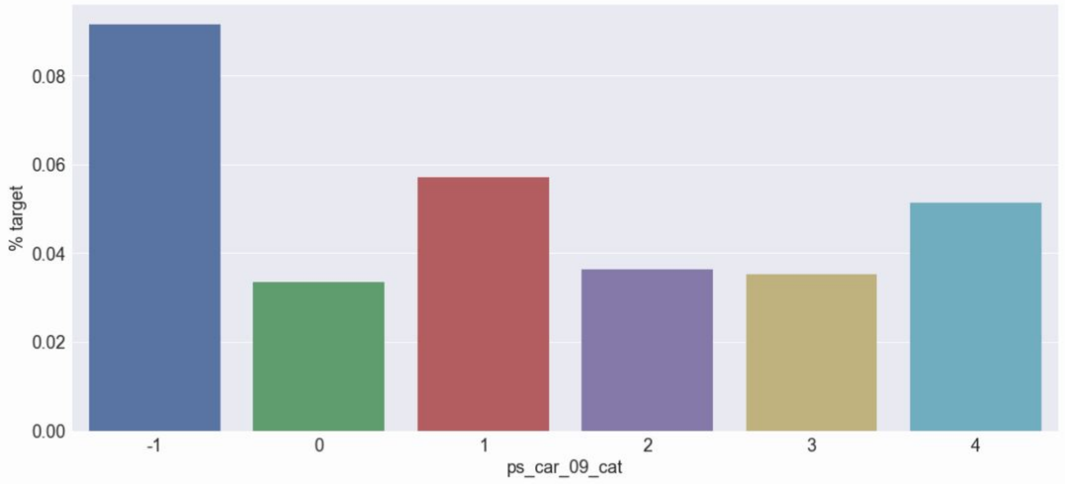


0.04
0.03
0.02
0.01
0.00
% target
0
1
ps_car_08_cat

```
<matplotlib.figure.Figure at 0x7f535175e240>
```

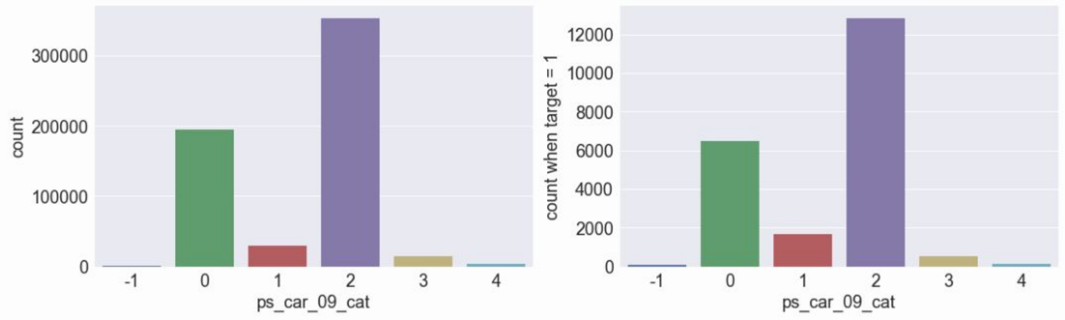


500000
400000
300000
200000
100000
0
count
0
1
ps_car_08_cat
15000
10000
5000
0
count when target = 1
0
1
ps_car_08_cat

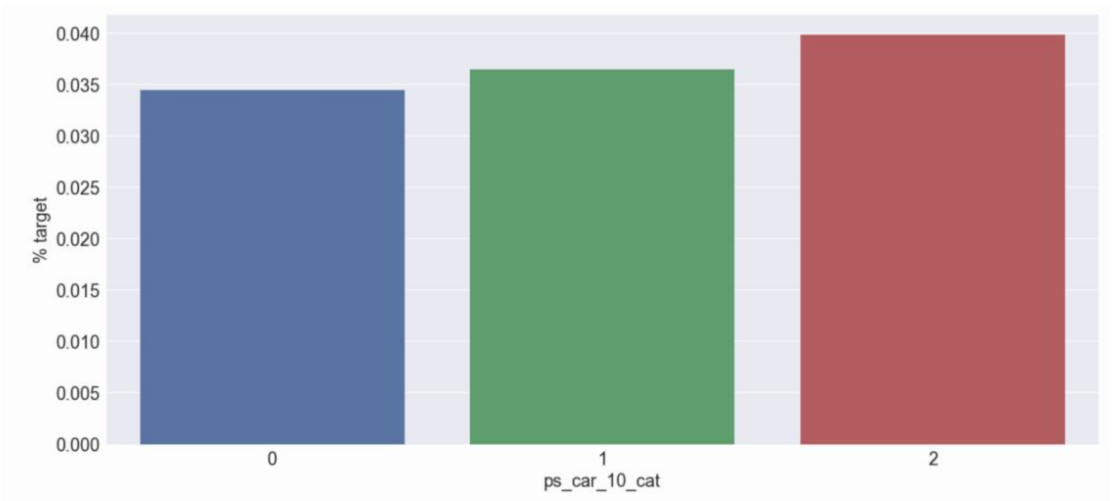


0.08
0.06
0.04
0.02
0.00
% target
-1
0
1
2
3
4
ps_car_09_cat

```
<matplotlib.figure.Figure at 0x7f535a03e320>
```

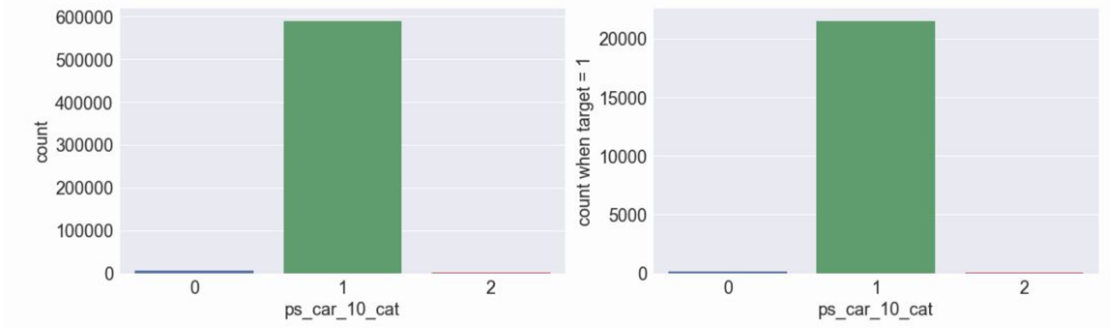


300000
200000
100000
0
count
-1
0
1
2
3
4
ps_car_09_cat
12000
10000
8000
6000
4000
2000
0
count when target = 1
-1
0
1
2
3
4
ps_car_09_cat

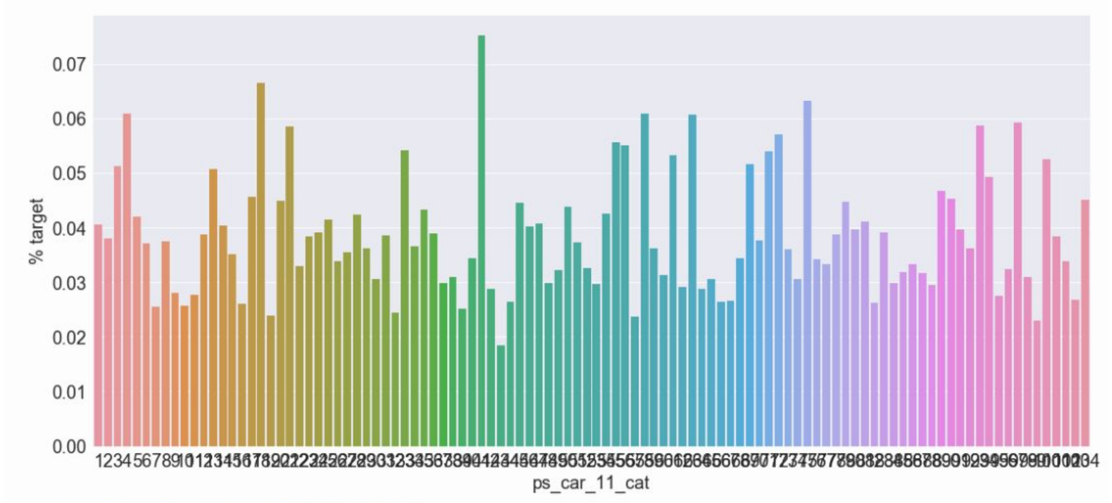


% target
ps_car_10_cat

```
<matplotlib.figure.Figure at 0x7f535a007438>
```

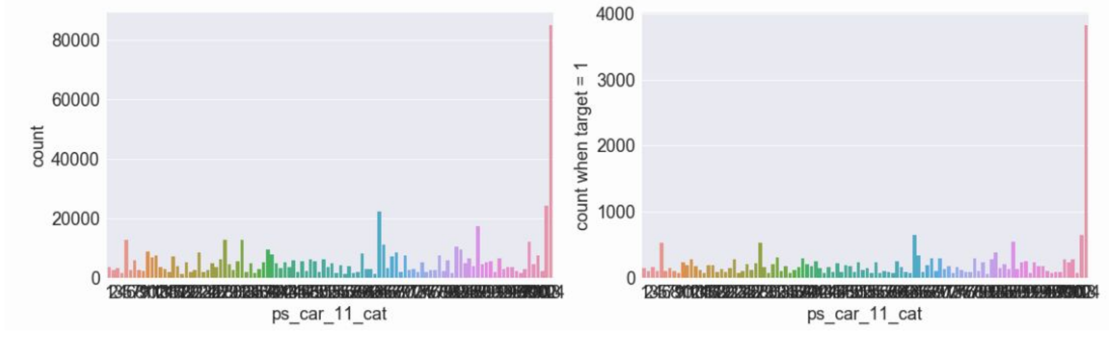


count
count when target = 1
ps_car_10_cat



% target
ps_car_11_cat

```
<matplotlib.figure.Figure at 0x7f53519bf400>
```



count
count when target = 1
ps_car_11_cat

We can get some useful information from those graphs:

1. The distribution of the categorical features is almost the same when we subset with *target = 1*.
2. There are many features where the probability of insurance claim is higher when the information is missing like, for example, ps_ind_02_cat, ps_ind_04_cat and ps_car_01_cat. It should be wise to use this information, instead of replacing it by the mode, for instance.

Algorithms and Techniques

There are many possible algorithms for classification problems. Some of them would be: Naive Bayes, Support Vector Machine, Decision Tree, Neural Networks. Scikit learn, a python library for machine learning, gives us many of them already implemented and ready to use. There is an interesting page where the classifiers are compared to each other: http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html.

Although we can discuss which algorithm would be better for each kind of dataset, it is also known that the fail-proof way to get the best algorithm for a problem is to test all of them. Well, that would take too much resources, indeed. At the same time, some changes in the dataset (like feature engineering, for instance) and hyperparameters tuning could also lead to differences between algorithms performance for that particular case.

This projects' goal is NOT to find the state of the art solution for that particular case, nor find the best algorithm for the Porto Seguro's dataset. Therefore, I have selected XGBoost, a gradient boosting algorithm. Some characteristics of XGBoost that makes it interesting for this particular dataset:

1. It can handle well imbalanced data, without need of other techniques;
2. It can model very complex data;
3. It can prevent overfitting by introducing several hyperparameters like, but not only, L1 and L2 regularization;
4. It has been used by many winners in kaggle competitions;
5. A plus, is that it minimizes our need of techniques preparing the data, like feature scaling, handling missing data, etc.

XGBoost seems to be a good fit for the problem discussed in this project, but, what finally led me to choose it was its proven performance in many kaggle competitions. Other implementations of boosting would also fit, for example, light GBM (Gradient Boosting Machine). We could also use neural networks, as they are very powerful (although sometimes hard to find a good setup), or a stack of different learners. There is no way we could say upfront which one would perform better, we have to implement them and compare the results.

As said before, our goal is not to find the best of them, but to explore XGBoost which is considered a very good and performant classifier for large datasets.

I'll give you now more details about XGBoost. The explanation below is based on the algorithm's introduction documentation. If you like check this out for more details: "Introduction to Boosted Trees".

XGBoost, the chosen algorithm for this project, is short of "Extreme Gradient Boosting", where the term "Gradient Boosting" is proposed in the paper Greedy Function Approximation: A Gradient Boosting Machine, by Friedman. It is used for supervised learning problems to try predicting a target variable y_i based on training data (of multiple features) x_i. There is also an implementation that can be used for regression problems.

The term *Gradient* comes from **"Gradient Descent"** which is the algorithm used to optimize the objective function (in general this is a function that describe the error for a particular model, and optimizing it often means minimizing the error function). In few words gradient descent, at each step, tries to adjust the parameters to get closer to the minimum.

The term *Boosting* is an **ensemble** (meta-)algorithm**.** Instead of relying on a single **weak learner** (which is a simple model) to predict the target variable, it combines many of them creating a **strong learner**. Boosting at each step chooses a new weak learner that better learns what was misclassified by the previous. XGBoost uses **Classification and Regression Trees (CART)** as weak learners, which, in simple terms, is a Decision Tree with real scores associated to its leaves.

Other techniques used by XGBoost:
- **Regularization**: "Regularization adds a penalty on the different parameters of the model to reduce the freedom of the model". That way, this technique is reducing variance and the resulting model should be less likely to overfit, which means better generalization.
- **Pruning**: "reduces the size of decision trees by removing sections of the tree that provide little power to classify instances". In other words, it tries to efficiently generate trees by "selecting" the best splits and discarding those that should not make the model better.

XGBoost is also a great algorithm because it has been implemented to be **fast**, **portable** and **distributed**.

XGBoost has many hyperparameters. Some of them are presented below. If you are interested, you can check all of them in the documentation. Most of the time there is no such thing as "right choice". Because of that, it is important to train and evaluate the model for each group of hyperparameters (by using Grid Search).

- **max_depth**: depth of a tree. If higher, more complex the model will be and more likely to overfit; Range: [0, INF]
- **min_child_weight**: minimum sum of instance weight (hessian) needed in a child. Range: [0, INF]

- **learning_rate**: control the step size. If too low, it can take too much time to converge. If too high, it can overshoot the minimum. Range: [0, 1]
- **n_estimators**: number of trees to compose the ensemble.
- **n_jobs:** number of parallel threads to be used. Can speed up processing.
- **gamma**: minimum loss reduction required to make a further partition on a leaf node of the tree. Range: [0, INF]
- **subsample**: subsample ratio of the training instance. Range: [0, 1]
- **colsample_bytree**: subsample ratio of columns when constructing each tree. Range: [0, 1]
- **reg_alpha**: L1 regularization parameter.
- **reg_lambda**: L2 regularization parameter.
- **missing**: value that should be considered as "missing" value.

To begin width XGBoost we may use default parameters, except for:

- missing: -1
- n_jobs: 4 (this is the number of logical processor in my computer)

The other parameters we might have to tune them later.

Benchmark

Using ROC AUC as scorer gives us, for start, one good starting point for benchmarking: 0.5 is what a completely random solution would achieve. In our dataset, always predicting the target as zero (the driver will NOT claim) also lead to a 0.5 score.

We can make this benchmark more interesting. I have trained a Gaussian Naive Bayes model with this dataset and got a score of 0.58. It is still a poor result, but slightly better than the previous one.

If you are interested on how I got that 0.58 score, you can get the source code and try yourself: https://github.com/rtpaulino/mlnd/tree/master/capstone.

III. Methodology

Data Preprocessing

As we will be using XGBoost, there are some preprocessing techniques that we **DON'T** need to do:

- Replace of missing values by the mean or the mode - that's not necessary as we can tell the algorithm that "-1" refers to missing data and it will handle it internally;
- Feature scaling: since XGBoost uses trees as base learners, feature scaling has no effect on the result;
- Label encoding: transform string categorical data to integer would be necessary, but we don't have that kind of data in our dataset.

We could make use of "One Hot Encode", but as Wang said in the quora post "Do we need to apply one-hot encoding to use xgboost?", that may be good or not - so it would be better to play around with that and see in which way we can get better results.

XGBoost deals well with unbalanced data, so we don't need to care about resampling or generating synthetic samples.

Implementation

The implementation to solve this problem was not hard at all. Actually, the data preparation is more difficult than training the model itself. First of all, we needed to import some libraries:

1. Data manipulation
   a. NumPy
   b. Pandas
2. Data Visualization
   a. Matplotlib
   b. Seaborn
3. SciKit Learn - Machine Learning Techniques
   a. Train Test Split
   b. Grid Search Cross Validation
   c. ROC AUC Score
4. XGBoost

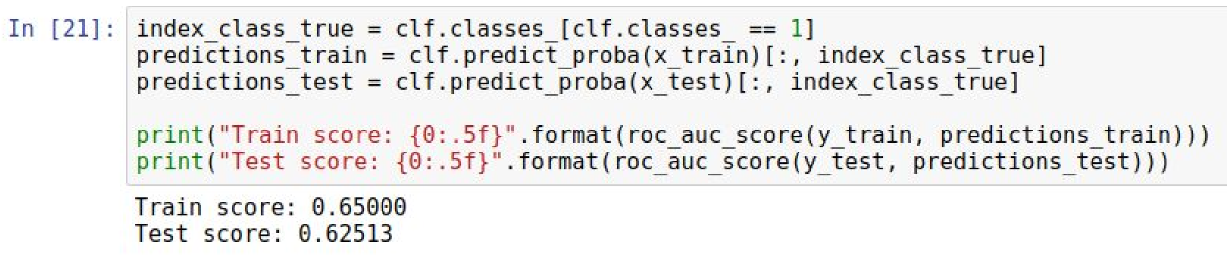
Below I outline the process I followed to get results using XGBoost:

1. **Split train and test data**: this is an essential part, so we can evaluate if the model is able to predict unseen data.
2. For each trained model, I will get the score using **roc_auc_score** (from sklearn). Most of them I'll get the score from train data set only. For the first, second and last model I will get also the score for test data. After that I will plot a graph showing the most important features used by XGBoost. This is possible because XGBoost uses trees as learners.

3. **First and Second Model**: as I have explained above, XGBoost does not require many data preparation as it internally handles it. To build my first models, I have just created an **XGBClassifier** with default parameters (except for n_jobs and missing) and trained it. The second model is similar except that, first, I transform the categorical feature using **One Hot Encode** (actually using pandas' get_dummies function).
4. The second model did perform better than the first, so all the next models trained will use the encoded features.
5. After training those first models, I'll **optimize XGBoost parameters**. I have based myself on the blog post "Complete Guide to Parameter Tuning in XGBoost (with codes in Python)". Most of the parameters were tuned using Grid Search technique.
    a. Set learning rate and number of estimators;
    b. Tune max_depth and min_child_weight;
    c. Tune gamma;
    d. Tune subsample and colsampe_bytree
    e. Tune alpha and lambda (regularization parameters)
    f. At last, reduce learning rate and maximize number of trees;

Using **XGBClassifier** was pretty straightforward. The only thing we had to pay attention was setting **eval_metric** to **auc** when fitting (training) the data and, when make predictions, use the method **predict_proba**, as we are interested in the probability and not the most probable label (target). As **predict_proba** return two values (as we have two classes), in our case we need to get the correct value (target = 1).

That way, we got the following results for our first model:

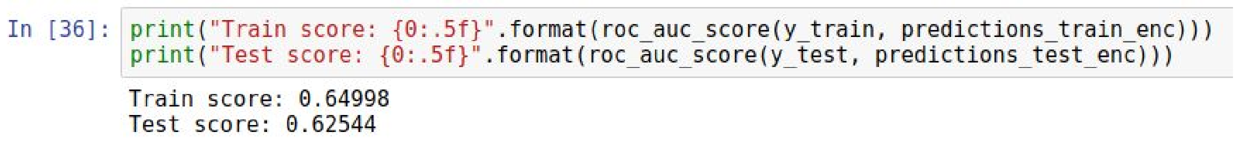


```
In [21]: index_class_true = clf.classes_[clf.classes_ == 1]
         predictions_train = clf.predict_proba(x_train)[:, index_class_true]
         predictions_test = clf.predict_proba(x_test)[:, index_class_true]

         print("Train score: {0:.5f}".format(roc_auc_score(y_train, predictions_train)))
         print("Test score: {0:.5f}".format(roc_auc_score(y_test, predictions_test)))

         Train score: 0.65000
         Test score: 0.62513
```

After encoding categorical features, we trained another model and got a bit better results:



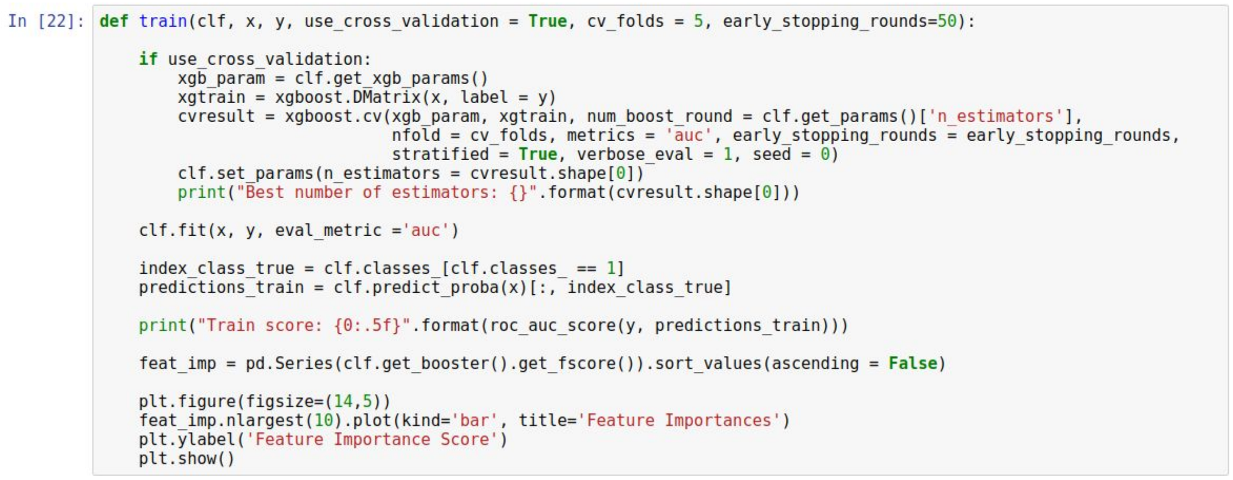
```
In [36]: print("Train score: {0:.5f}".format(roc_auc_score(y_train, predictions_train_enc)))
         print("Test score: {0:.5f}".format(roc_auc_score(y_test, predictions_test_enc)))

         Train score: 0.64998
         Test score: 0.62544
```

We can see that the train score got a little bit worse, but the test score got a bit better. Because of that, I have decided to use only the encoded features in the optimization phase.

Refinement

To proceed with fine tuning our base model, I have written a function to train the model, print the score and plot the feature importance. It is almost the same I have done before, but, I can make use of the XGBoost's "cv" (from Cross Validation) method to help us estimating the best number of trees (n_estimators).



```
In [22]: def train(clf, x, y, use_cross_validation = True, cv_folds = 5, early_stopping_rounds=50):

             if use_cross_validation:
                 xgb_param = clf.get_xgb_params()
                 xgtrain = xgboost.DMatrix(x, label = y)
                 cvresult = xgboost.cv(xgb_param, xgtrain, num_boost_round = clf.get_params()['n_estimators'],
                                       nfold = cv_folds, metrics = 'auc', early_stopping_rounds = early_stopping_rounds,
                                       stratified = True, verbose_eval = 1, seed = 0)
                 clf.set_params(n_estimators = cvresult.shape[0])
                 print("Best number of estimators: {}".format(cvresult.shape[0]))

             clf.fit(x, y, eval_metric ='auc')

             index_class_true = clf.classes_[clf.classes_ == 1]
             predictions_train = clf.predict_proba(x)[:, index_class_true]

             print("Train score: {0:.5f}".format(roc_auc_score(y, predictions_train)))

             feat_imp = pd.Series(clf.get_booster().get_fscore()).sort_values(ascending = False)

             plt.figure(figsize=(14,5))
             feat_imp.nlargest(10).plot(kind='bar', title='Feature Importances')
             plt.ylabel('Feature Importance Score')
             plt.show()
```

*NOTE: I have set **stratified** to true when using CV method, as our data is unbalanced.*

When using the CV method, it is possible to set "early_stopping_rounds", which means that, if the algorithm cannot improve the results after that number of rounds, it should stop! At each iteration, using cross validation, XGBoost add one more tree. That way, we can get the best value for n_estimators. Of course, that can be different when the learning rate is different. So, how do we proceed? The lower the learning rate, more tree we will need and, in consequence, more time to train and evaluate.

So, first we set a learning rate to some value (I have set to 0.3) and, by using CV method explained above, I get the best number of trees; in our case 100. For my computer, that seemed to be good start point to start searching for better parameters.

The problem about using Grid Search is processing time. If at the same time you try to test, for example, two parameters with 3 values each, and using 3 folds for cross validation, it will be need 27 fits to get the results. At the first time, each fit took around almost 2 minutes, what mean I would get almost 1 hour to run this process only one time!

As I did not have that plenty of time nor wanted to pay for a better machine to do the heavy work, I found a particular interesting parameter in XGBoost: **tree_method**. That is "the tree construction algorithm used in XGBoost" (reference). By default it was using "exact", but when set to "hist" it took only **20 seconds/fit.** As that could lead to different results (when comparing with the first models I have trained), I used "hist" when using Grid Search and, after that, I have used the default value to get the results and compare to other models. The best reference I found about histogram method was this issue on github: https://github.com/dmlc/xgboost/issues/1950.

For each group of parameters I tuned, I often had to run two or three times with different possible values search for best parameters. It was an interesting approach to try improving one or two feature at a time. If we would try to grid search with all of them, it would take too much time as the number of fits would increase drastically.

There is a reason behind the combinations of parameters chosen to tune together. *max_depth* and *min_child_height* for example directly influence how the trees will be constructed and when there will be a split; *gamma* is related to pruning; s*ubsample* and *colsample_bytree* reduce overfitting by using sampling. *Lambda* and *alpha* are related to regularization.

After some grid searches and parameter tuning, I have trained 6 new models:

Name	**Parameters**	**Train Score**	**Test Score**

clf3	learning_rate = 0.3
	n_estimators = 100
		0.68061	-

clf4	max_depth = 3
	min_child_weight = 5
		0.67058	-

clf5	gamma = 0.3	0.67058	-

clf6	colsample_by_tree = 0.8
	subsample = 0.9
		0.66221	-

clf7	reg_alpha = 13
	reg_lambda = 0.2
		0.66340	-

clf8	learning_rate = 0.1
	n_estimator = 5000
	(use CV to get best n_estimators, which was 3291)
		0.66794	0.63195

IV. Results

Model Evaluation and Validation

As result of this project, we have come to a tree-based model trained using XGBoost with the following hyperparameters:

- Max_depth: 3
- Learning_rate: 0.01
- N_estimators: 3291
- Gamma: 0.3
- Min_child_weight: 5
- Max_delta_stop: 0
- Subsample: 0.9
- Colsample_bytree: 0.8
- Colsample_bylevel: 1
- Reg_alpha: 13
- Reg_lambda: 0.2
- Scale_pos_weight: 1
- Base_score: 0.5
- Missing: -1

What can impress in this model parameters is the number of trees (3291) and the use of lots of parameters to avoid overfitting, like gamma, subsample and regularization terms.

The resultant model got a ROC AUC score of **0.66794** for the training set and **0.63196** for the test set. As we have used many techniques to prevent overfitting, the difference between training and testing scores is acceptable.

Justification

The problem we are trying to solve is really hard. If it was easy, auto insurance companies could just really charge those driver that WILL for sure be involved in car accidents. The goal of this project is to get better predictions and we achieved that successfully:

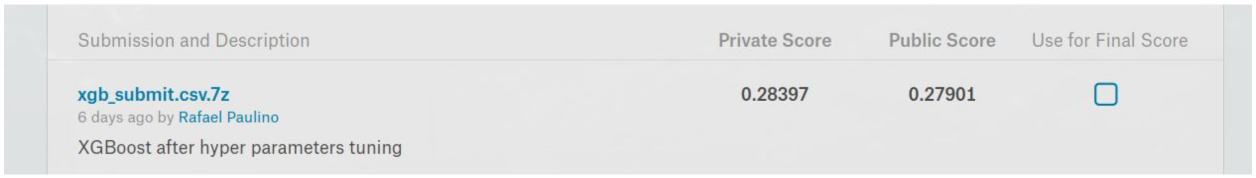
Benchmark (Naive Bayes) model score: 0.58
XGBoost model score: 0.63 (8.62% better)

As ROC AUC score ranges from 0.5 to 1, we could normalize that metric from 0 to 1:

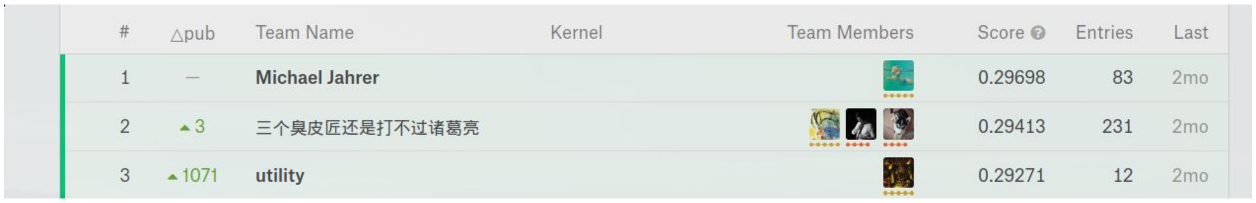
- Benchmark: (0.58 - 0.5) * 2 = 0.16
- XGBoost: (0.63 - 0.5) * 2 = 0.26 (62.5% better)

Our model is significantly better than our benchmark model. But, the benchmark model is still a very poor model. A score of 0.63 for ROC AUC is still considered a poor model. That means there still room for improvements.

For curiosity, as this project was based on a kaggle competition, I have submitted the results to see how good would it perform when comparing to other competitors:

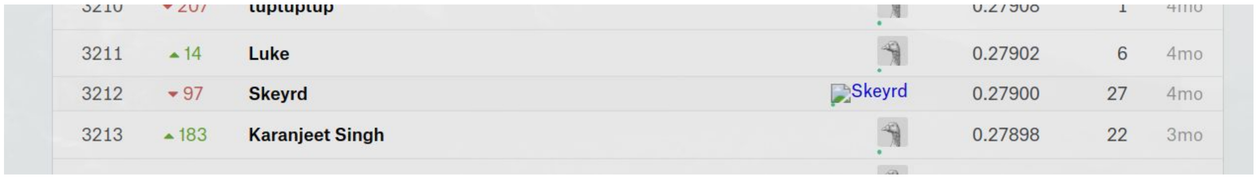


Submission and Description	Private Score	Public Score	Use for Final Score
xgb_submit.csv.7z 6 days ago by Rafael Paulino XGBoost after hyper parameters tuning	0.28397	0.27901	☐



#	△pub	Team Name	Kernel	Team Members	Score	Entries	Last
1	—	Michael Jahrer			0.29698	83	2mo
2	▲3	三个臭皮匠还是打不过诸葛亮			0.29413	231	2mo
3	▲1071	utility			0.29271	12	2mo

That score would lead to 3212 position out of 5169 competitors. (62%)



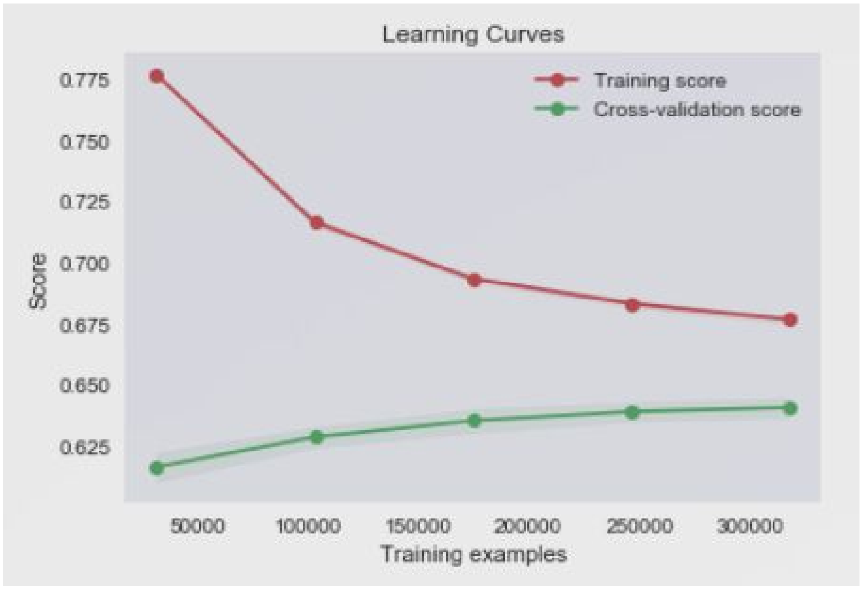
3211	▲14	Luke		0.27902	6	4mo
3212	▼97	Skeyrd	Skeyrd	0.27900	27	4mo
3213	▲183	Karanjeet Singh		0.27898	22	3mo

V. Conclusion

Free-Form Visualization

The dataset provided by Porto Seguro is very good. It can be used as a playground to test performance of many algorithm and/or techniques and compare with different solutions, at least for that kind of classification problem (very unbalanced dataset). In this project I have worked with XGBoost, but it would be nice to compare results with other approaches. We could also use that dataset to compare the impact of feature engineering. What kind of technique can lead to better performance? Is that improvement considerable or not? At the same time, this project can be a good start point to anyone who wants to learn about XGBoost.

Below, let us analyze the learning curves of model proposed:



1. It seems that even with a larger number of test data, this model would not provide us much better performance. For example, between the last two points of the green curve we could not see much improvement with 50 thousand more training samples.
2. We can also see that the training and cross-validation curves are getting close to each other. That means our model is not overfitting.

Can I say then that XGBoost solved the problem? Yes, it has. But in this kind of problem, I don't think that the real question is that. Perhaps, a better question would be: does this solution is better than what we have got so far? For companies like Porto Seguro, the idea is to improve predictions so they can leverage better services with best prices and, of course, that would lead to higher profits. Is the model of this project better than Porto Seguro's? Probably not! There is still room for improvements as I'll discuss some of them below.

Reflection

The process for this project can be summarized using the following steps:

1. Problem statement and dataset took from kaggle competition;
2. Created a naive bayes classifier as benchmark;
3. Selected XGBoost as algorithm to create a model;
4. Verified that only One Hot Encoding was "possibly" needed for preprocessing;
5. Splitted train and test datasets;
6. Created first model with default parameters;
7. Performed several trainings using Grid Search to fine tune hyperparameters;
8. Scored the final model with test set.

One challenge I have faced in this project was computational time when fine tuning. Performing Grid Search and Cross Validation are heavy and sometimes one optimization took over an hour. As I didn't intended to take too much time on that (and often it is needed), I have reduced the possibilities for Grid Search (using maximum 9 combinations) and reduced CV folds (only 3). It would be better to use 5, for example, but it would be much slower. Another thing that helped a lot to speed up the tuning process was the use "hist" as *tree_method*. That drastically reduced one fit from 2 minutes to 20 seconds.

Improvement

Sure there is room for improvements! It would be possible to better fine tune taking more time on that. But often, fine tuning only improves performance by small amount. Doing only that would not lead me to the first page of ranking in the kaggle competition.

There are at least two things that could be done better for significant improvement:

1. **Transform dataset**. Often, in machine learning, you get better results by working on the data itself. Are the calculated features useful? I have found some "kernels" on kaggle's that said those feature would be of no help. There were also some that created new features by combining two of them; some discarded features; and, at last, some "smoothed" categorical features. I have seen a post from the leader on the competition and he had used unsupervised learning to create new features by generating "noise-free" features.
2. **Combining models**. That's also a great approach to get better results. We could not only use XGBoost, but combine XGBoost with other classifiers. All models have its strengths and weaknesses; by combining models, we can use the strengths of each model.