

# Machine Learning Engineer Nanodegree

## Capstone Project

Paulo Roberto de Oliveira Castro August 1st, 2018

### I. Definition

#### Project Overview

Insurance is a data-driven business. Since it's beginning, it's possible to observe[1] that the area was always focused on gathering information about a situation that involves risk (driving a car, having a house, etc.) and then measuring that risk, so it's possible to profit by assuming someone's risk and getting payed for that in exchange.

However, especially in Brazil, this is a rather expensive business for the final costumer. Buying new car, even if you're a very good driver, means a huge insurance spending. Sometimes it even prevents the costumer from buying it at all. Therefore, a good prediction about what is the real risk for a specific client means that the provider can charge a more reasonable amount of money for their insurance products. Machine Learning is therefore, one of the ways to achieve this[2]

This project has origin on a Kaggle competition[3], from where we take a dataset provided by the insurance company Porto Seguro with dozens of features for more than five hundred thousand clients, alongside with a flag if that client claimed or not[4].

#### Problem Statement

In this project we try to solve the problem of telling apart costumers that drive safely from those who don't based on the information held at the insurance company. To accomplish this, we will use several classification techniques from Machine Learning, including feature engineering and data cleaning steps.

First, we will explore the dataset and establish a benchmark solution, then proceed screening several classification algorithms. Then we will explore several approaches for data cleaning and feature engineering, measure their impact on the final result. After that we will also try to optimize the hyper-parameters of the chosen algorithms. After that, we will produce an ensemble of those models, such that the final answer overfits less the training set.

The expected outcome of this project is a model ensemble, featuring several classification models and some data preprocessing steps. This kind of solution

is widely used in Kaggle, being applied in most of the competition winners' solutions.

## Metrics

The performance of our solutions will be assessed using the Gini Coefficient[5], which can be computed as:

$$Gini = 2 * AUC - 1$$

Where AUC is the area under the ROC curve as a proportion of the total area (which is one). In practice this means that the Gini Coefficient is the proportion of the area under the ROC curve and the total area, but only considering the region above the random line in the ROC. The Gini Coefficient of 1 indicates a perfect model, and of zero indicates one that performs as well as random guess. Less than that, the model performs worse than random guess. The area under the ROC curve is a very common way of measuring performance of binary classification tasks[6], and the Gini Coefficient is a simple linear transformation of it.

The advantage of using this metric is that we have a clear benchmark, below which the answer is unacceptable (random guess), and also we would be able to measure our performance accounting for the target distribution on the test set. If the target distribution is highly skewed, metrics like accuracy would give an almost perfect score to solutions that simply guess the most common class. On the other hand, when using the Gini Coefficient we simply compute the false/true positive rates[7], which are insensitive to class imbalance.

## II. Analysis

### Data Exploration

As mentioned before, we use the dataset provided by Porto Seguro, containing 57 features for 595212 clients, alongside with a flag if that client claimed or not. The data is anonymous: we can't neither get any information about the client (apart from an ID) nor from the features, since all of them are encoded and names in a way that it's impossible to guess its meaning.

Some information, however, is still present in the feature names. Features has part of the name indicating whether it's a individual features (with the string "ind" in the middle of the name), a feature about the car ("car") or a calculated feature, made by Porto Seguro based on other features ("calc"). Feature engineering can be hard, specially when we don't have the true meaning of the features, but we should be careful when using this kind of feature, as it

may only introduce noise to some of our models (specially non-linear ones, that can discover those relations as part of the learning process).

Another information present in the names of the features in our dataset is its type. The suffix of the name indicates whether a feature is categorical (suffixed with “cat”), binary (“bin”) or numeric (no suffix, and can be either a floating point or an integer feature). This raises another concern that we may address in our solution: categorical features (specially non-binary ones) often require some form of encoding – if we encode these variables as numbers, we are implying order between the categories, which may not be the case.

A sample of the dataset is shown in the following table:

Missing values is another concern we should have about this particular dataset. Some categorical and numerical data have missing values, with a total of thirteen features presenting this kind of issue. The number of missing values seems to vary (from a few percent up to more than 70% in one of the columns). There is some structure, however, in the way that the missing value work. The presence of missing values in one of the features may reduce or increase the chance that we see a missing value in other column. With all these properties, the way we will handle missing values may be of crucial importance for some algorithms.

When looking on the target distribution, we also find another characteristic to deal with in our problem: the dataset is highly unbalanced. The number of people how claims the insurance is much lower than the number of those who don't (only ~4% of the people claim). This unbalance means that we may see some problems with some algorithms. For example, techniques that has as loss function that don't distinguish errors from the most common and the less common target class may result in a model that only guess the most frequent target value. In our case, we would only miss ~4% of the value, but the Gini Coefficient would be low, since we never guess the least frequent target value correctly. To deal with this we could try and apply balancing techniques on the algorithm level or by under/super-sampling the original training set.

The distribution of the variables was also investigated. We looked for some

ill-shaped distributions, containing outliers, as well as co-linear variables that could be redundant. However, when we look at the variables and their joint distributions, we found only 11% of the examples were outliers in at least one feature (univariate outlier, and these examples could not be outliers when considering all variables at once). As of the joint distributions, we didn't find any specially co-linear variable. When dealing with linear methods, we shall remember this, as we could deal with this kind of situation by using a different loss function such as some regularization technique as adding the L1 norm to our loss function. However, when looking for correlation between numerical columns, we found that some of the variables (for example `ps_car_12` and `ps_car_13`) have high positive correlation while some (like `ps_ind_14` and `ps_ind_15`) have high negative correlation.

When inspecting the joint distribution between the variables and the target, we found that a significant number of variables have different distributions depending of the target, but the difference is always very weak. This means that we shouldn't expect this classification problem to be easily solved. Those small effects should be more easily dealt with by using some non-linear technique as boosting or bagging.

## Exploratory Visualization

First, as a manner to easily detect patterns in our data, we inspect the joint distribution of our variables. Taking a look at the numerical variables, we can see that most of them are conditionally independent, however we can detect dependencies in pairs like '`ps_calc_10`' and '`ps_calc_08`'. Those are indications that we have a high mutual information between those pair of variables, and that knowing one could be enough to detecting a pattern about the target.

Another way of detecting such redundancies is plotting a correlation matrix between those variables. This will give a more objective way of measuring the redundancies. As we can see, some variables are redundant and probably would add little to no information about the target on our models.

Another important information when modelling our problem is the distribution and amount of missing data we have on this dataset. We can't know if the missing data is missing at random, or the missingness have a meaning, but we can at least inspect its distribution before deciding the approach, specially since the models we tried handle missingness in different way.

The first plot is the distribution of missing data throughout the dataset. Here we can see that we have a lot of features without missing data, but a bunch of them do. We highlight `ps_car_03_cat` and `ps_car_05_cat`, that are missing for most of the customers.

The data completeness column indicates that those missing values are most likely not missing at random, since most of the times the customer have not one, but

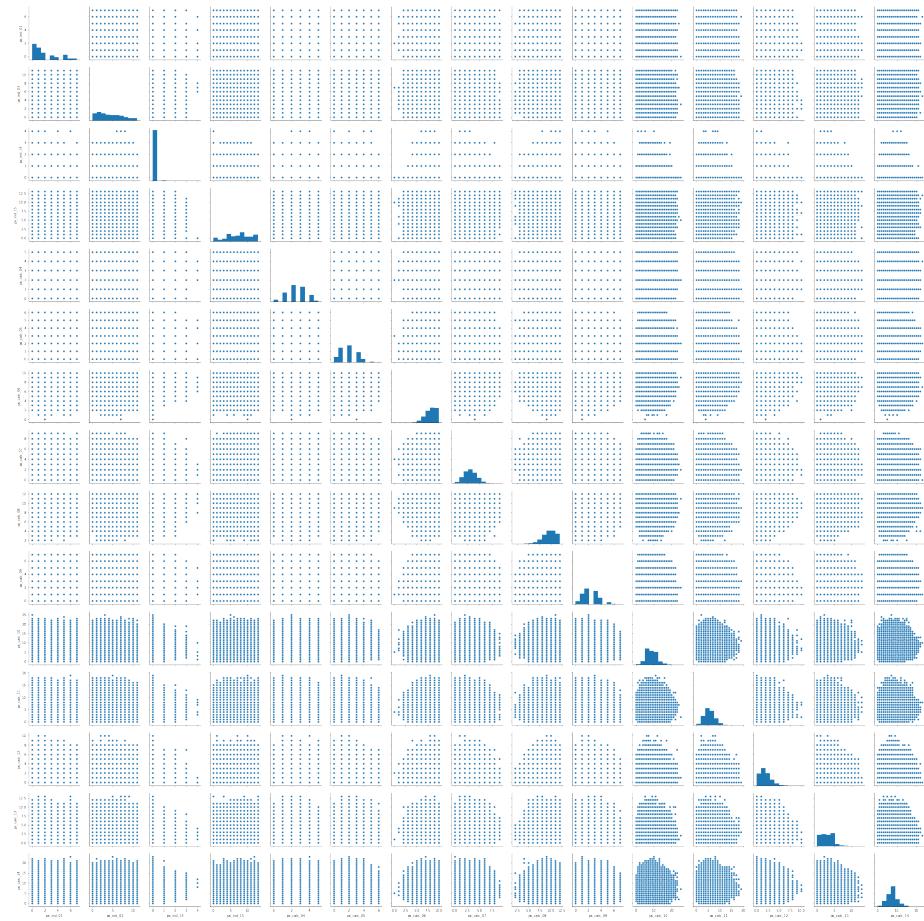


Figure 1: categorical features distributions

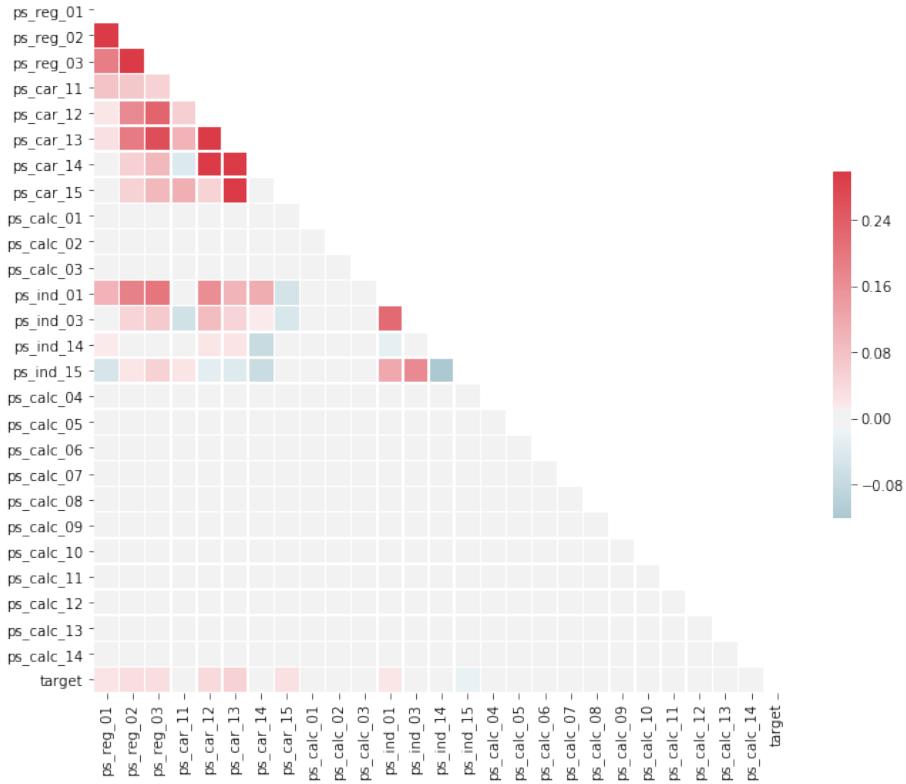


Figure 2: correlation matrix

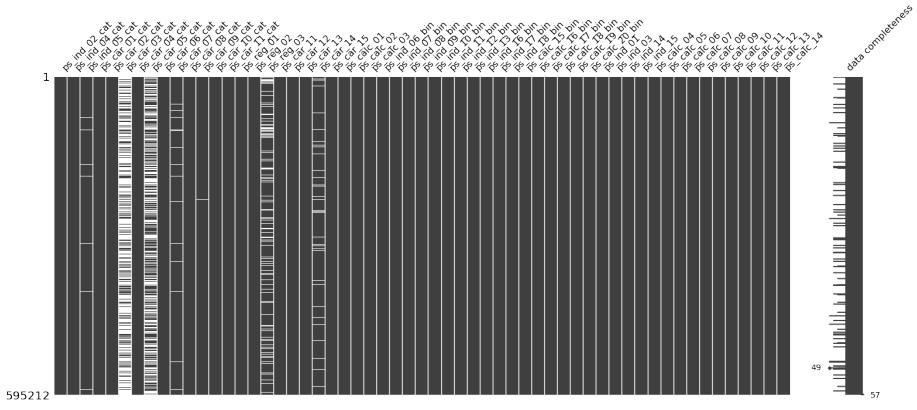


Figure 3: missing distribution

several missing features at once.

Another important aspect to analyze about this problem is the joint distribution between features and targets. When we inspect those distributions we can see that on some variables (e.g. `ps_calc_01`) we have a clear difference between the distribution of the feature values when the class is positive and when the class is negative.

Inspecting this is important since we can have an insight about whether the problem is easy (most variables have very different distributions) or not. In this case, those relations are more nuanced.

We could have done the same plots for the categorial distribution, only to arrive at the same kind of conclusion.

## Algorithms and Techniques

This is a classic classification problem, and therefore, several classification algorithms can be used to solve it. The most basic one is the Logistic Regression. This linear model will use linear combinations of the features to predict the probability of each of the two target classes. While simple, it's very fast to train and will detect any simple linear pattern in the dataset. The logistic model models the probability of a sample  $i$  as:

$$\text{logit}(\mathbb{E}[Y_i | \mathbf{X}_i]) = \text{logit}(p_i) = \ln\left(\frac{p_i}{1 - p_i}\right) = \boldsymbol{\beta} \cdot \mathbf{X}_i$$

The algorithm basically works by finding the maximum likelihood for the samples, that is, maximizing  $p(y|\beta, x)$ . Also, it's very easy to interpret, since we can tell what happens to the target when changing one of the variables. When training a logistic regression, we are also able to add a regularization term to its loss function. If we use the L1 regularization term, the solution is most likely to be sparse. On the other hand, if we use the L2 term, we only force that each of the parameters of the linear model must be small, but not necessarily zero. On the other hand, the L2 regularization results in more robust solutions than the L1 regularization, while being more computationally efficient.

However, as we noted in the previous sections, a linear model shouldn't be able to explain the relation between the target and the features. Therefore, we will also test several non-linear models, such as boosting, bagging, neural networks and ensemble of many models.

Boosting models consist of iteratively learning weak models (usually decision trees) and then combining those weak models in a stronger learner. One example of this is the gradient boosting (implemented in XGBoost and LightGBM for example) in which each weak learner will fit the residual  $y - F(x)$  of the current strong learner  $F$ . The optimization step is such that we compute the residuals for

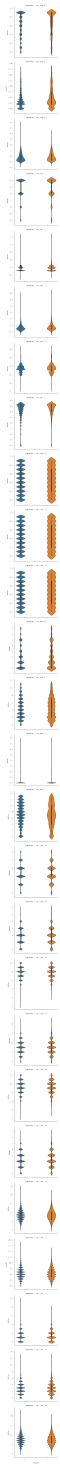


Figure 4: target joint distribution with numerical features  
8

each sample, fit a new base learner  $h$  to the residuals and compute the multiplier  $\gamma$  so that:

$$\hat{F} = \sum_m \gamma_m h_m(x) + const$$

Another boosting technique used in this project is the AdaBoost, in which each weak learner are tweaked in favor of examples missclassified by previous learners. The optimization algorithm is similar to gradient boosting, but using an exponential loss  $\sum_i e^{y_i f(x_i)}$ .

Those models are nonlinear, since they use decision trees and ensemble techniques as learners, therefore they could better capture the patterns in our dataset. Also, this techniques effectively reduce bias (and variance) and can be better in generalizing to the test set.

Neural networks are non-linear models comprised of several layers, each computing linear combinations and using a non-linear function of top of that to compute its output:

$$\sigma(\sum_i w_i x_i) = y_i$$

The coefficients of those linear combinations are optimized buy computing the gradients of a loss, in an hierarchical manner, and then descending the gradient. This minimizes the error, and achieves a local optima that tends to be similar to the global one. Neural networks achieved state of the art performance on machine learning problems related to sound and image classification, and play important role in new reinforcement learning techniques.

On the neural network side, we use entity embedding [7] to encode our categorical variables, allowing for fast neural network training. This technique was previously applied to classification problems and was able to reach third place in a competition with very simple features[8].

Other algorithms will also be tested, like Random Forest and Support Vector Machines (RBF kernel). These models not only could improve our metrics alone, but could also be used in the final ensemble to improve generalization. The final model will be an ensemble of several of those models. This could reduce the variance and improve generalization a little bit, and is usually used in Kaggle competitions.

## Benchmark

As discussed earlier, the Logistic Regression is the most simple and interpretable linear model we can use for this sort of problem. Also, it is usually used by

insurance companies while dealing with this kind of problem. Therefore, it will be our algorithm of choice for our benchmark.

However, this algorithm doesn't deal with missing values. Therefore we need to apply a very simple scheme to fill missing data. In this case, we will use the median of each feature in the training set to fill those values up. Other techniques will be tested in time.

The result we obtained was a Normalized Gini of -0.24, which means that it is worse than random guess. As a comparison, if we predict only good drivers, we reach a Normalized Gini of 0.005. To reach this result, we used the Logistic Regression implementation from the python library scikit-learn, using the default parameters. We used 70% of the dataset as training data, and 30% as test.

While bad, this result isn't much surprising. This problem was extracted from a Kaggle competition from an insurance firm, that usually uses this kind of technique. Therefore this should be a dataset to which Logistic Regression wasn't easily applicable, but a highly problematic one (so things would get interesting on the competition).

### III. Methodology

#### Data Preprocessing

To reach our best result with Boosting, we first discard features calculated by Porto Seguro, which our non-linear algorithms must be able to figure out by themselves. This will reduce the amount of features, reducing the variance. We also applied a target encoding to our categorical features [9], since encoding them as numbers add a false sense of ordering to the categories. This method was found to be slightly better than a simple one-hot encoding (as we can see in notebook #06)

After that, to each numeric feature we add a corresponding boolean one that states whether that feature is above or below the mean. We also create the same set of variables using the median as the reference value. Those values will simplify the encoding of the numeric features, simplifying the job of the decision trees (since they can now split on this boolean feature instead of guessing what value to split on).

To deal with missing data we could use some strategies. We could use the missing data as is (as supported by XGBoost and LightGBM implementations of Gradient Boosting) or fill the missing values with mean or median of the training set. After some tests (that can be found on notebook #05), we find that the gradient boosting performed better by filling the missing values with the median of the training set, so this is what we applied in our final Boosting algorithm.

The dataset is highly unbalanced with respect to the target labels. To deal with this some balancing techniques were tested, such as undersampling, oversampling

(SMOTE) and class weights, but none lead to an increase of the Gini metric (see notebook #04).

For the entity embedding neural network, we used a label encoder to pre-encode the categorical features into integers, so that they would enter the neural network in the correct format. Also, we excluded categorical variables with only 2 or less possible values. After that, we only needed to ensure that the variables followed the same order defined in the neural network.

## Implementation

All the work, both experimentation and final models, were implemented in Jupyter notebooks using the Python language. This allows for flexibility to experiment and iterate over ideas in a fast manner, while keeping the report style clear and concise. Each test, from the exploratory analysis to the final model, was implemented in a separate notebook, and all of them are available alongside this report.

All data wrangling was done using the libraries Pandas (data loading, dealing with missing values and feature creation), scikit-learn (for transformations) and imbalance-learn (for dealing with the unbalanced target). Also, all the machine learning algorithms we used are available in scikit-learn, Keras (with Tensorflow backend), LightGBM and XGBoost libraries.

The metric used in this project wasn't available in any of those libraries. Therefore, a new implementation had to be provided. We used the same API used by scikit-learn metrics `gini_normalized(actual, pred)`. This allows for integration with all python libraries, which expect this kind of function signature. The implementation used in this work follows the one provided in this kernel[10]. This implementation is very fast and easy to read.

We also had to implement target encoding[9]. We used the concept of scikit-learn's transformers, and implemented this encoder as one. To use it, we can simply use the `fit` and `transform` methods on our data. As a possible complication, in this implementation, we only accept pandas's `DataFrame` as input, so it isn't fully compatible with Scikit Learn framework (alough it is close to that).

For the entity embedding part, following this implementation[11], we chose to implement the neural network using Keras. For each categorical feature, we built an embedding layer with a smaller dimensionality than the number of possible values for that categorical variable (e.g. if the variable had 10 possible values, we built an embedding with less than 10 dimensions). This was done using the `Embedding` layer from keras. For the other variables, we used simple `Dense` layers of 16 dimensions output. All of those were combined in a `Merge` layer and used as imput to a neural network of 4 layers with dropout. Since we are dealing with a binary classification problem we used the binary crossentropy as loss function and Adam as our optimizer given its fast convergence speed and

overall result close to the best on several problems. The neural network was trained in a low-end GPU (NVIDIA GeForce 830M) in less than one hour.

## Refinement

First (as seen in notebook #3) we tried several algorithms contained in the Python library `scikit-learn`. We used the classifiers `RandomForestClassifier`, `KNeighborsClassifier`, `MLPClassifier`, `GradientBoostingClassifier`, `AdaBoostClassifier` and `SVC` from `sklearn`, all with default parameters. The `lightGBM` and `XGBoost` libraries' classifiers were also used with their default parameters. The result is that gradient boosting was indeed the best way to deal with this dataset, giving us the highest Normalized Gini (0.2700) when compared to algorithms like AdaBoost (0.2565), Random Forest (0.069), Multi-layer Perceptron (0.2407) and K-Nearest-Neighbors (0.0277).

For the boosting algorithm, we started with 0.2700 of Normalized Gini, without dealing with missing data, and also without categorical encoding or balancing the dataset. We proceeded to find the best way to deal with the unbalanced target, just to find that all of the techniques we tried (under-sampling, over-sampling with SMOTE or class weights) had no positive impact on the performance (notebook #4). After experimenting with techniques of dealing with missing values, we found that filling with the mean would raise the result to 0.2718 of Normalized Gini (notebook #5).

We then focused on the encoding of the categorical features (notebook #6). We experimented with One-Hot Encoding, which had little to no impact on the models performance (possibly because of the way XGBoost deals with categorical features), but Target Encoding raised the performance up to 0.2728 of Normalized Gini.

We proceeded to check whether feature normalization would affect performance (notebook #7), and found that neither Standard Scaling, Robust Scaling, Max-Abs Scaling nor Min-Max Scaling lead to any improvements on this particular dataset. The same can be said about dimensionality reduction algorithms like PCA and truncated SVD (notebook #8).

As the last step, we performed the implementation of a bagging ensemble model, so that our best solutions would be combined to create a better model. This was done by simply weighting the NNs, XGBs and Gradient Boosting scores with 0.4, 0.4 and 0.2 respectively. This tuning was done by either hand and by using a logistic regression on top of the scores. However, those hand-made values ended-up having the best score on the test set. The improvement with this technique was sensible, increasing predictive power while reducing overfitting.

## IV. Results

### Model Evaluation and Validation

After a series of Hyper-Parameter optimization steps for several models, we discovered that the best model for our problem is an ensemble of a Neural Network with an embedding layer (as described in the implementation section) with a Gradient Boosting model. The final neural network used in this is comprised of the embedding layer (with a dense layer with 16 neurons), followed by 4 dense layers with 80, 20, 10 and 1 neuron respectively. All but the last layer had `relu` activation and a dropout rate varying from 0.35 for the first layer to 0.15 of the other two hidden layers. We used binary crossentropy as loss, and optimized with the ADAM algorithm. The Gradient Boosting was performed by using both the `scikit-learn` default implementation and the XGBoost library (all with default parameters as stated above). The ensemble was performed with a bagging algorithm, with the weights being decided by hand to be 0.4 for the NN, 0.4 for XGBoost and 0.2 for the `GradientBoostingClassifier`.

Using this setup, we arrived at a mean Normalized Gini of ~0.28 across several runs with different train/test splits.

This indicates that this specific model is robust, since changes to the train/test-set choices didn't affect the results that much. The result is a model that has a good generalization power, and tends to have the same performance on unseen data.

Those two models (Neural Networks and Gradient Boosting) were chosen for the ensemble since they were very performant individually, but also because they are rather different models in the way they model the posterior distribution. This means that, when one of the models is bad at identifying a pattern on a particular region of the feature space, the other model could perform much better. Ensemble models are less prone to overfitting, since all of our models would have to agree before predicting the class of a data point.

It's important to see that the ensemble of the model was extremely important to reach such performance. No model alone could reach more than 0.275 consistently. However, by combining models that don't have much correlation between them, we can improve our predictive power while increasing our generalization capacity.

### Justification

The model proposed in this work can be considered better than the benchmark when looking at the proposed validation metrics suggested by the authors of the Kaggle competition. The benchmark (a simple Logistic Regression) wasn't able to perform better than random guess, with a negative Normalized Gini, while the ensemble model was able to reach a Normalized Gini of 0.27, which indicates reasonable predictive power.

Both results are so different that is reasonable to affirm that the benchmark was beaten and, therefore, the purpose of this work was accomplished.

## V. Conclusion

### Free-Form Visualization

As a form of visualizing our results we are going to see how our predictions look like in the test data. Since our problem problem is binary, the predictions can be represented in what is called a confusion matrix: we plot a matrix with the number of occurrences in each quadrant, defined by the `true` label of the example and the `predicted` label.

In this case, we have a matrix that is highly skewed for the first quadrant, which is the `true` label 0 and the `predicted` label zero. Although this is a correct prediction, we don't want the model to predict always the same label. Luckily, in this case, we have a few cases of `true` label 1 and `predicted` label also 1, meaning that in some cases our model was able to make a correct positive prediction.

### Confusion Matrix

It is also important to notice that this matrix was generated based on a choice of threshold, above which we consider the prediction score to result in positive label. In our case, we chose 0.5 just to exemplify the kind of prediction we make with this model, but when deploy a model like this we usually choose the threshold based on a business metric, such as the return of money that choosing a certain threshold would give us. Defining such a metric is beyond the scope of this project, but is certainly necessary to use this kind of model.

Also on the visualization of our results, an important metric to have is usually the importance of the features. We can see which features are more or less important and, based on that, we can decide whether it is worth it to keep or throw away a certain feature and what would be the impact of doing that on the model's performance.

In the case of the XGBoost model, one of the most important model of our stack, we can use the default implementation of feature importance to evaluate that. The metric used in this case (the default `weight` metric in XGBoost library, here called the `F score`), show how many times a tree from our boosting algorithm was split using this variable. Therefore, the higher the score, the more important is the variable.

While there are many other ways to measure the importance of a variable on an XGBoost model (using `gain`, `cover` or even libraries like `shap`), this metric is easy to grasp and usually enables the user to decide what feature are more or less important for the model.

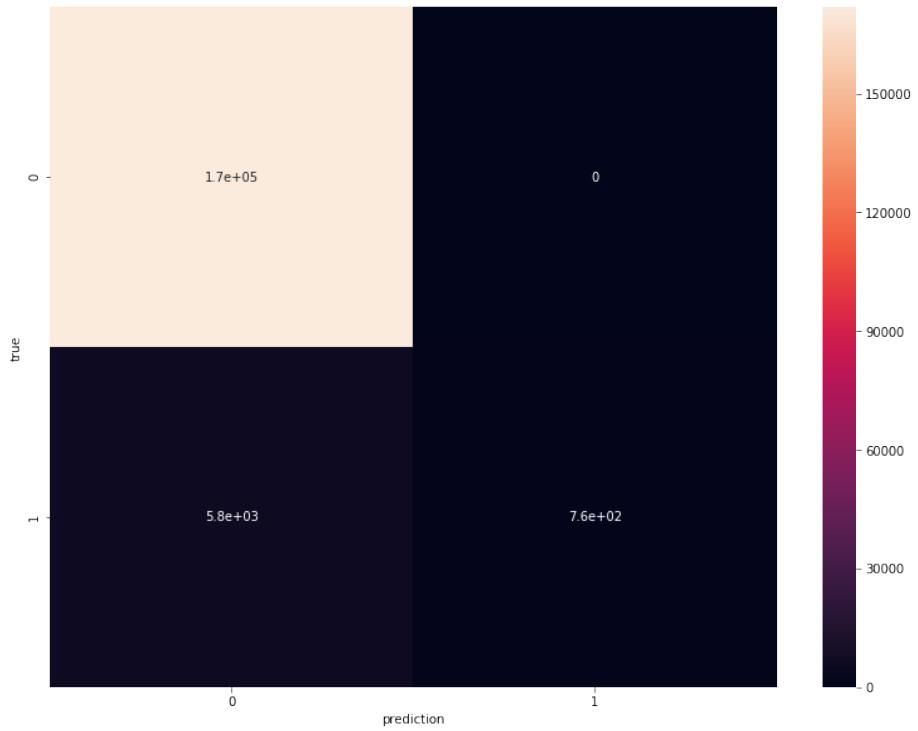


Figure 5: Confusion Matrix

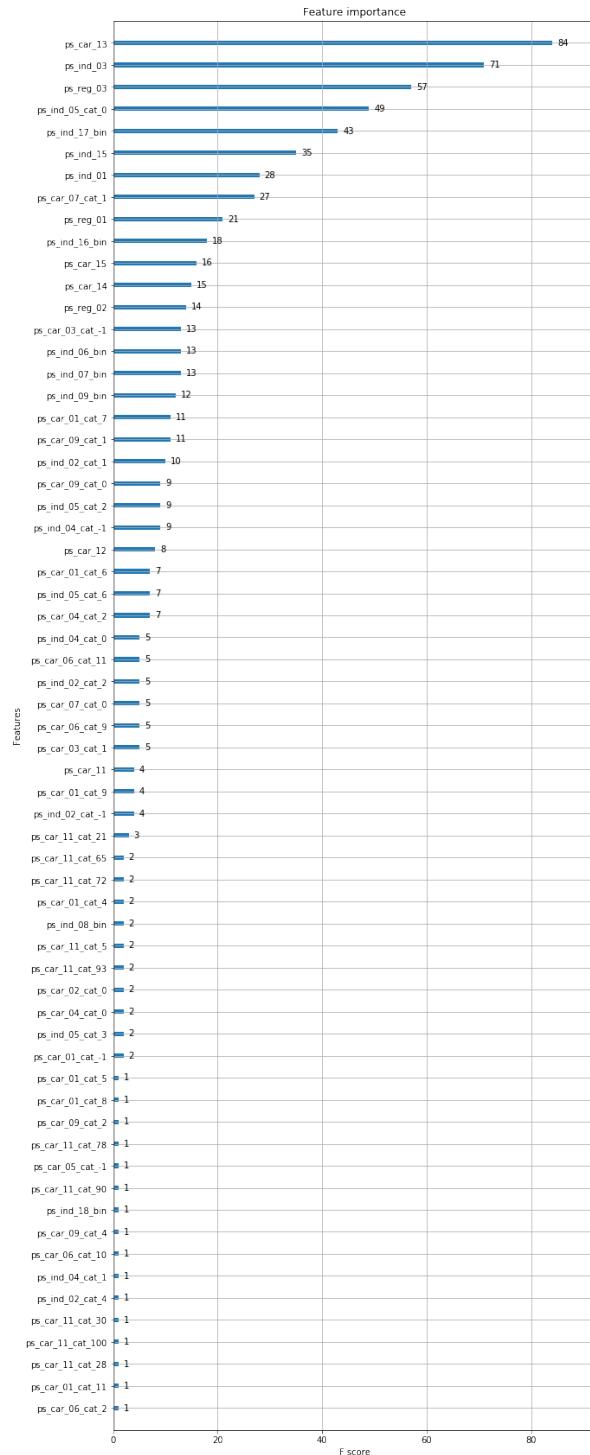


Figure 6: Feature Importance  
16

## Reflection

In this work, we analyzed the problem of predicting a driver will initiate an auto insurance claim in the next year, based on information held by an insurance company. To accomplish this, we started with data exploration and benchmark models (logistic regression, which we use for comparison) to get a sense of the problem. Next we surveyed several well-known machine learning algorithms for this task and also applied several techniques for feature engineering and pre-processing. In this step we retained the best solutions for our problems so that we could build an ensemble model with the two best approaches: a neural network with an embedding layer and a gradient boosting model.

Difficult aspects of this problem are the target imbalance and the categorical nature of a large number of variables. In the first problem, we perceived that the imbalanced nature of the target didn't help very biased classification models since they could optimize accuracy by predicting the most common class. To circumvent this, we both used higher-variance models (such as Gradient Boosting and Neural Networks) and resampling techniques such as under sampling and SMOTE for over sampling. Those techniques are what enable us to reach higher Normalized Gini performance for this dataset.

On the second problem, we observed that a large number of features are categorical, but with a simple representation of those categories, the classification algorithms always have a hit performance-wise. The solution is to try to encode those variable so that we don't convey order between the classes while maintaining a less-redundant representation of the data. This is where the embedding network helped a lot. This model optimized the representation of the categorical features for the specific classification problem, leading to a simple and elegant solution with slightly better performance than the Gradient Boosting model, but without the high amount of feature engineering that this solution requires.

This kind of algorithm can be used whenever we have a bunch of categorical features on an imbalanced dataset that a simple linear model can't be used for prediction. In special, our expectations for this solutions were met, although a larger distance between performances was expected.

## Improvement

This work could be improved by going into hyper-parameter optimization more thoroughly: more parameter combination for the Gradient Boosting model, which is very slow and could take days or even weeks to optimize, but also the optimization of the neural network architecture. Adding more layers, performing more kinds of regularization, changing the objective functions and increasing training epochs could all lead to better performance of this model.

The representation of the categorical features could also be worked on. As the neural network model showed, this step is fundamental to the performance of

the model. One possibility is using the representation found on the embedding layer of the neural network as input of other kind of model.

Another source of improvement could be in the ensemble of those models. The model weights could probably be improved, and other models could be also added to the mix, such as different neural networks and other kinds of boosting techniques. The last thing we could mention is the access to the meaning of those variables, so that our cleaning process can be enhanced and we can get better insights on the patterns we can find in this data.

As seen in the Kaggle competition, many better solutions exist, but all public information on those solutions leads us to think that the general idea of solving this problem is basically the same that we applied in this work.