EE 660 – Machine Learning from Signals: Foundations and Methods

Porto Seguro’s Safe Driver Prediction

Yixin Sun, Ben Zhang

yixinsun@usc.edu, benzhang@usc.edu

December 3, 2017

Abstract

Car insurance is a very important protection for drivers and it takes more concern with the growing number of cars. Today, auto insurance companies are using machine learning techniques to make claim predictions. High accuracies reduce the cost for good drivers and raise the price of insurance for bad drivers. Therefore, an accurate model needs to be studied to predict the probability that a driver will file an auto insurance claim. In this report, gradient boosting and logistic regression are discussed to approach this problem. Feature engineering, missing data imputation, feature extraction and model selection are also performed. Key results are that logistic regression requires more data preparation work to achieve high performance, and gradient boosting is more appropriate for complex problems.

1. Problem Statement and Goals

The main goal is to predict the probability of a driver not being a safe driver, or the probability that a driver will initiate an auto insurance claim in next year [1]. This is a classification problem, and instead of predicting a binary label, e.g. {good driver, bad driver}, the probability can give an idea about if the driver is a potential bad driver or not.

Some great challenges include high dimensionality, sparsity, and possibly nonlinear relationship. For example, very few are bad drivers, which leads to an imbalanced data and poses an obstacle to high performance. There is also a large number of missing values. Consequently, imputation needs to be completed before some learning algorithms are applied.

2. Related Work

None.

3. Project Formulation and Setup

3.1 Gradient boosting

The algorithm Yixin Sun used was gradient boosting. Below is a description of gradient boosting, followed by the reason why it would perform well on this dataset.

Boosting is a greedy algorithm that trains a simple classifier, or base classifier, by a 'weak learner' at each iteration to only perform better than chance, and fit all base classifiers together to give the final estimated target function. Boosting fits adaptive basis function models and can be represented in the form of equation (3.1).

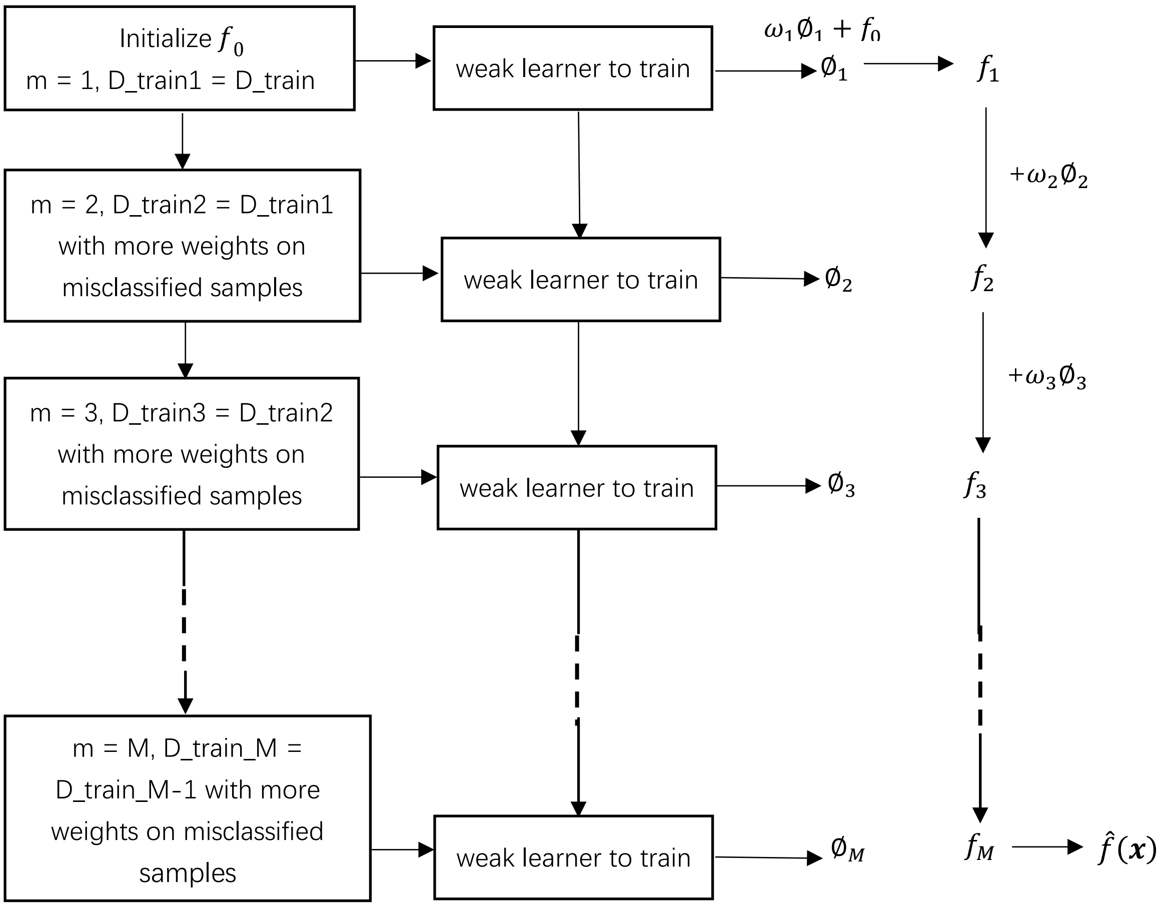
$$\hat{f}(\boldsymbol{x}) = f_0 + \sum_{m=1}^{M} \omega_m \emptyset_m(\boldsymbol{x}, \boldsymbol{\theta_m}) \tag{3.1}$$

where $\emptyset_m$ denotes the mth trained classifier generated by the weak learner, and ω_m is the importance of the base classifier.

Boosting works by applying the weak learner sequentially to weighted training data, with more weights on samples that were misclassified by earlier base classifiers [2]. Typically, the weak learner is a one-stage decision tree model resulting in a split of a feature. The split is chosen to minimize a cost function $\sum_{i=1}^{N} L\,(y_i, f(\boldsymbol{x_i}))$. Specifically at mth iteration, the goal is to solve

$$\omega_m, \boldsymbol{\theta_m} = \underset{\omega, \boldsymbol{\theta}}{argmin} \sum_{i=1}^{N} L(y_i, f_{m-1}(\boldsymbol{x_i}) + \omega\emptyset(\boldsymbol{x_i}, \boldsymbol{\theta})) \tag{3.2}$$

and then set $f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + \omega_m \emptyset(\boldsymbol{x}, \boldsymbol{\theta_m}))$. Once the algorithm stops at Mth iteration, the estimated target function $\hat{f}(\boldsymbol{x})$ is equal to $f_M(\boldsymbol{x})$. The flowchart for boosting is shown below.



Initialize f_0
m = 1, D_train1 = D_train
weak learner to train
$\omega_1\emptyset_1 + f_0$
$\emptyset_1$
f_1
m = 2, D_train2 = D_train1 with more weights on misclassified samples
weak learner to train
$\emptyset_2$
$+\omega_2\emptyset_2$
f_2
m = 3, D_train3 = D_train2 with more weights on misclassified samples
weak learner to train
$\emptyset_3$
$+\omega_3\emptyset_3$
f_3
m = M, D_train_M = D_train_M-1 with more weights on misclassified samples
weak learner to train
$\emptyset_M$
f_M
$\hat{f}(\boldsymbol{x})$

For gradient boosting, the cost function $\sum_{i=1}^{N} L\,(y_i, f(\boldsymbol{x_i}))$ is defined to be $\sum_{i=1}^{N} |y_i - f(\boldsymbol{x_i})|$. To minimize the cost function, gradient descent is applied. The gradient at step m for the ith sample is computed first:

$$g_{im} = \left[\!\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]\!\right]_{f(x_i) = f_{m-1}(x_i)} \tag{3.3}$$

and then $\boldsymbol{\theta_m}$ is updated using equation (3.4) below.

$$\boldsymbol{\theta_m} = \underset{\boldsymbol{\theta}}{argmin} \sum_{i=1}^{N} (-g_{im} - \emptyset(\boldsymbol{x_i}, \boldsymbol{\theta}))^2 \tag{3.4}$$

3.2 Why gradient boosting

As discussed early, the challenges are: high dimensionality, possibly nonlinear relationship, missing data, and imbalanced classes. Gradient boosting can solve these potential problems effectively. Features form a high dimensional space, and this is even worse after one hot encoding. Gradient boosting typically uses the decision tree model as the weak learner, which automatically chooses the best feature to split at each iteration. After the algorithm, gradient boosting can aggregate how many times a feature was chosen, thus giving the feature importance. Features that are not important will not be selected, and thus reducing the dimensionality.

Another advantage of gradient boosting is that it can generate non-linear decision boundary, thereby solving the possibly non-linear problem.

In terms of missing data, gradient boosting feeds a sample into both directions of a split of that missing feature, and computes the cost function. If the left child node minimizes the cost function more, then this missing data sample goes to the left child node; otherwise, it goes to the right child node. Therefore, gradient boosting can handle missing data well and is a good fit for this dataset.

3.3 XGBoost and parameters to tune

Now the only challenge is the imbalance of data. In this report, a gradient boosting library called XGBoost is utilized. This library provides a hyperparameter, scale_pos_weight, which balances the positive and negative weights. If this parameter is tuned well, XGBoost can deal with imbalanced data. Easy ensemble method is also tried to solve the imbalance, which improves performance slightly. Details will be presented in 5.4. Table 3.1 shows the parameters to tune in this project for XGBoost, including a brief introduction to these parameters.

Parameters name	Introduction to parameters
eta	Learning rate or step size for gradient descent. It can shrink the feature weights to prevent overfitting [3]. Typically 0.01-0.2.
num_boost_round	Number of rounds for boosting. This is the M in equation (1).
gamma	Minimum required cost function reduction to split a node. It prevents overfitting.
max_depth	The maximum depth for a decision tree. The larger, the more complex the model. Typically 3-10.
min_child_weight	Minimum required sum of weights in a child. A large value prevents overfitting while a too small value might lead to underfitting.
subsample	The percentage of samples to use for growing trees. This is similar to the ratio of samples to use in bagging methods. It brings randomness to the model and prevents overfitting. Typically 0.5-1.
colsample_bytree	The percentage of features to use for growing trees. This is similar to the ratio of features to use in bagging methods. It brings randomness to the model and prevents overfitting. Typically 0.5-1.
lambda	L2 regularization term. The larger, the more conservative the model.
alpha	L1 regularization term. The larger, the more conservative the model. Useful in case of high dimensionality.
scale_pos_weight	Balance the positive and negative weights. Useful in case of imbalanced data.

Table 3.1 Parameters to tune for XGBoost

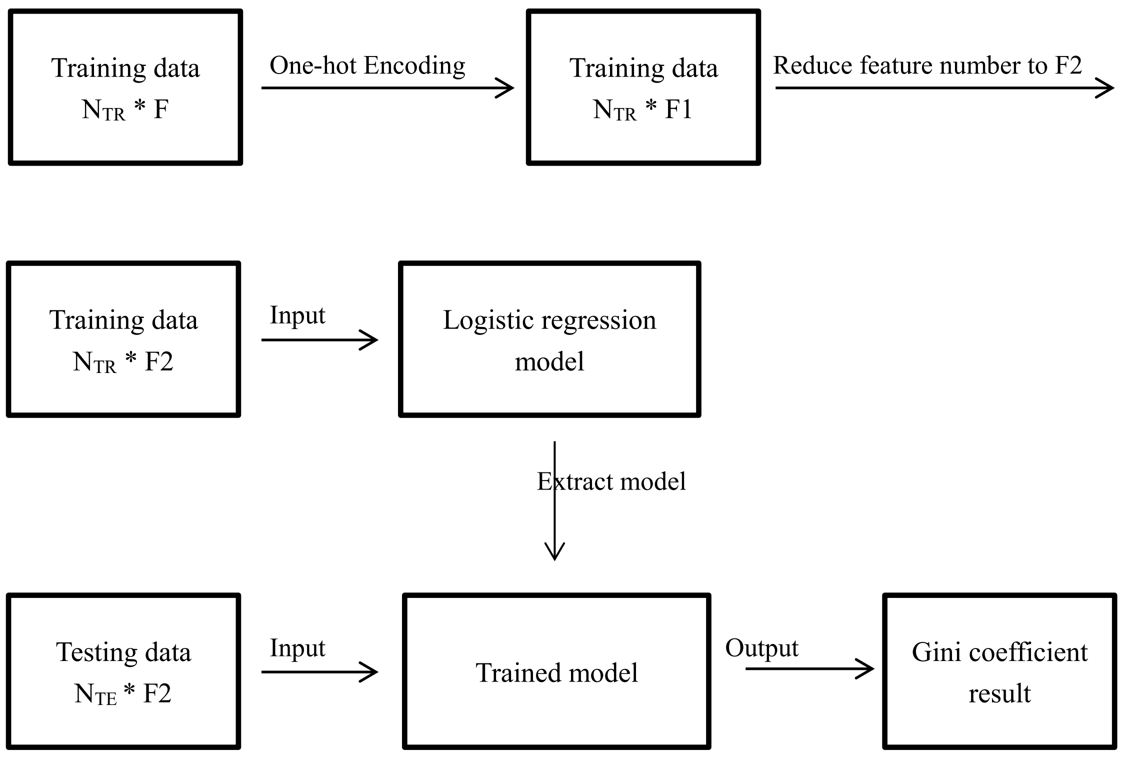
3.4 Logistic regression

Ben Zhang selected logistic regression model. It is easy to train logistic regression model to predict the probability of labels. Besides, the assumption is that each data sample is i.i.d. distributed.

To prepare the dataset for logistic regression, all categorical features, such as *car_brand,* are transformed into binary features using one-hot encoding. This step expands feature space. Next, feature extraction is performed, because high dimensionality increases problem complexity, takes long training time, and leads to bad performance for logistic regression. Three methods are applied:

Principle Component Analysis (PCA), Non-negative Matrix Factorization (NMF) and Independent Component Analysis (ICA). Details will be illustrated in 5.3.

A brief flow chart is below:



Logistic regression algorithm:
Consider a two-outcome probability space [4]:

$$P(0) = p$$
$$P(1) = 1 - p$$

The logit function is

$$z = \log\left(\frac{p}{1-p}\right)$$

Then, the logistic function becomes

$$p = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

For a sample point **x** in feature space, project it onto ω to convert into a real number z in range $-\infty$ to $+\infty$.

$$z = \omega_0 + \underline{\omega} \cdot \underline{x} = \omega_0 + \omega_1 \cdot x_1 + \omega_2 \cdot x_2 + \cdots + \omega_d \cdot x_d$$

Map z to the range 0 to 1 using the logistic function

$$p = 1/(1+e^{-z})$$

Optimization method: stochastic gradient descent is used to optimize ω, so the model gives best possible reduction of training set label [5]. For each iteration t from 1 to T_epoch (50 or 100),

ω_0(t+1) = ω_0(t) + learning_rate * (y(t) - yhat(t)) * yhat(t) * (1 - yhat(t))

For each i from 1 to d (total d features),

ω_i(t+1) = ω_i(t) + learning_rate * (y(t) - yhat(t)) * yhat(t) * (1 - yhat(t)) * x_i(t)

until some stop condition is satisfied.

If learning rate is too small, it will cost much time to converge. If learning is too large, it may jump over the optimal solution. Thus, the learning rate should be selected carefully. Finally, the predicted yhat is derived by

$$\text{yhat(t)} = 1 / (1 + e^{-(\omega_0 + \underline{\omega(t)} \cdot \underline{x(t)})})$$

Logistic regression is used for this dataset because it is computationally cheap to obtain a training model and is resistant to overfitting by trying different regularization methods. It is also intuitively clear to interpret weights as indicators of features importance from the model. However, since logistic regression only fits well for linear boundary, it serves as a comparison for gradient boosting.

4. Methodology

In this report, the hypothesis set consists of $\hat{f}(\boldsymbol{x}) = f_0 + \sum_{m=1}^{M} \omega_m \emptyset_m(\boldsymbol{x}, \boldsymbol{\theta_m})$ and $\hat{f}(\boldsymbol{x}) = sigmoid(\boldsymbol{\omega}\emptyset(\boldsymbol{x}))$, corresponding to gradient boosting and logistic regression model respectively. The reason for constructing this hypothesis set has been introduced in the previous section.

After setting the hypothesis set, the issue becomes how to utilize the data. Since Kaggle has already provided training set and test set, there is no need to split data. Also, because there are many hyperparameters, the training data is not split into training and validation but is used for 5-fold cross-validation. Here, 5-fold is applied because the dataset suffers overfitting according to some kernels on Kaggle, which means that larger K-fold may lead to higher chances to overfit.

To sum up, 595213 samples are used for training, 892816 samples are for testing, and 5-fold cross-validation is used to tune parameters.

The next step is feature preprocessing. Features consists of continuous, ordinal, categorical, and binary features. One-hot encoding is performed on categorical features. Some features that have 0 values in the correlation matrix and very small feature importance are directly removed to reduce the problem complexity. New features are also created to improve the predictive power. Details will be covered in 5.1 and 5.2.

To sum up, there are 57 variables before feature preprocessing and 203 variables afterwards, which is reasonable in terms of 595213 training samples, although the dimensionality is very high.

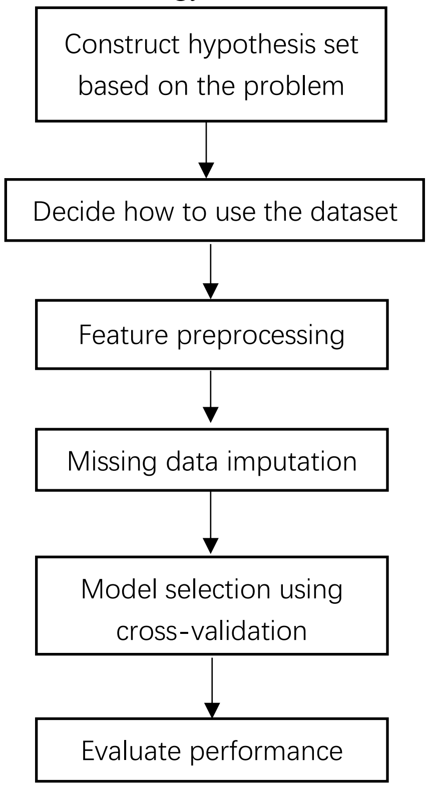
Afterwards, missing data is imputed. An important note here: since the training set has a very large size, it is impractical to impute missing data at each cross-validation loop due to computation time. Therefore, imputation is conducted before cross-validation, and labels are ignored in case of cheating.

Detailed analysis is covered in 5.3.1 and 5.3.2. The main strategy is using KNN, random forest classifier and random forest regression to predict the missing values.

Training and cross-validation is now brought into picture. At this step, the main purpose is tune parameters in table 3.1, such that a final model can be selected based on the validation performance. For logistic regression, it is important to reduce the dimensionality; thus, PCA, NMF, and ICA are tried at each cross-validation step.

At last, the final model is trained using the entire training set and parameters selected in CV. Then the model is evaluated on test data. The evaluation metric is Gini coefficient due to the problem domain.

The flowchart below shows the entire methodology.


Construct hypothesis set based on the problem
Decide how to use the dataset
Feature preprocessing
Missing data imputation
Model selection using cross-validation
Evaluate performance

5. Implementation

5.1 Feature space

Due to privacy, all features are encrypted by the auto insurance company, and the actual meaning of them is thus unknown. However, they have properties in common and can be grouped. Features starting with 'ps_ind' provide individual information. Some might be age, while some might be gender. Features starting with 'ps_car' provide car information, possibly including car's brands. Features starting with 'ps_reg' give the location information of the individual or car, and 'ps_calc' features are pre-calculated variables based on some information such as manufacture year. In brief, there are four types of features, namely individual, car information, region information, and manually calculated variables.

Also, features whose name ends with '_bin' are binary. Features whose name ends with '_cat' are unordered categorical, and for those whose name ends with neither '_bin' nor '_cat', they are either ordered categorical, or continuous real number. Table 5.1 presents the information of these features.

Feature	Feature type	Data type	Range or cardinality
ps_ind_06_bin, ps_ind_07_bin, ······, ps_ind_13_bin, ps_ind_16_bin, ps_ind_17_bin, ps_ind_18_bin; ps_car_08_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	binary	int	0, 1
ps_ind_02_cat	categorical	int	1, 2, 3, 4, NaN
ps_ind_04_cat, ps_car_02_cat, ps_car_03_cat, ps_car_05_cat, ps_car_07_cat	categorical	int	0, 1, NaN
ps_ind_05_cat	categorical	int	0, 1, 2, 3, 4, 5 ,6, NaN
ps_car_01_cat	categorical	int	0, 1, 2, 3, 4, 5 ,6, 7, 8 ,9 ,10, 11, NaN
ps_car_04_cat	categorical	int	0, 1, 2, 3, 4, 5 ,6, 7, 8, 9
ps_car_06_cat	categorical	Int	0, 1, 2, ···, 16, 17
ps_car_09_cat	categorical	int	0, 1, 2, 3, 4, NaN
ps_car_10_cat	categorical	int	0, 1, 2
ps_car_11_cat	categorical	int	cardinality 104
ps_ind_01, ps_calc_09	ordinal	int	0, 1, 2, 3, 4, 5, 6, 7
ps_ind_03	ordinal	int	0, 1, 2, ···, 10, 11
ps_ind_14	ordinal	int	0, 1, 2, 3, 4
ps_ind_15	ordinal	int	0, 1, 2, ···, 12, 13
ps_reg_01, ps_calc_01, ps_calc_02, ps_calc_03	ordinal	float	0.0, 0.1, 0.2, 0.3, ···, 0.8, 0.9
ps_reg_02	ordinal	float	0.0, 0.1, 0.2, ···, 0.9, 1.0, 1.1, ···, 1.8
ps_reg_03	continuous	float	0.061237~4.037945, NaN mean = 0.894047, std = 0.345413
ps_car_11	ordinal	int	0, 1, 2, 3, NaN
ps_car_12	continuous	float	0.1~1.264911, NaN mean = 0.379947, std = 0.058300
ps_car_13	continuous	float	0.250619~3.720626 mean = 0.813265, std = 0.224588
ps_car_14	continuous	float	0.109545~0.636396, NaN mean = 0.374691, std = 0.045610
ps_car_15	continuous	float	0.0~3.741657 mean = 3.065899, std = 0.731366

Table 5.1 Detailed information of features

For this dataset, features starting with 'ps_calc_' are dropped. This can be illustrated by the correlation heatmap in figure 5.1.

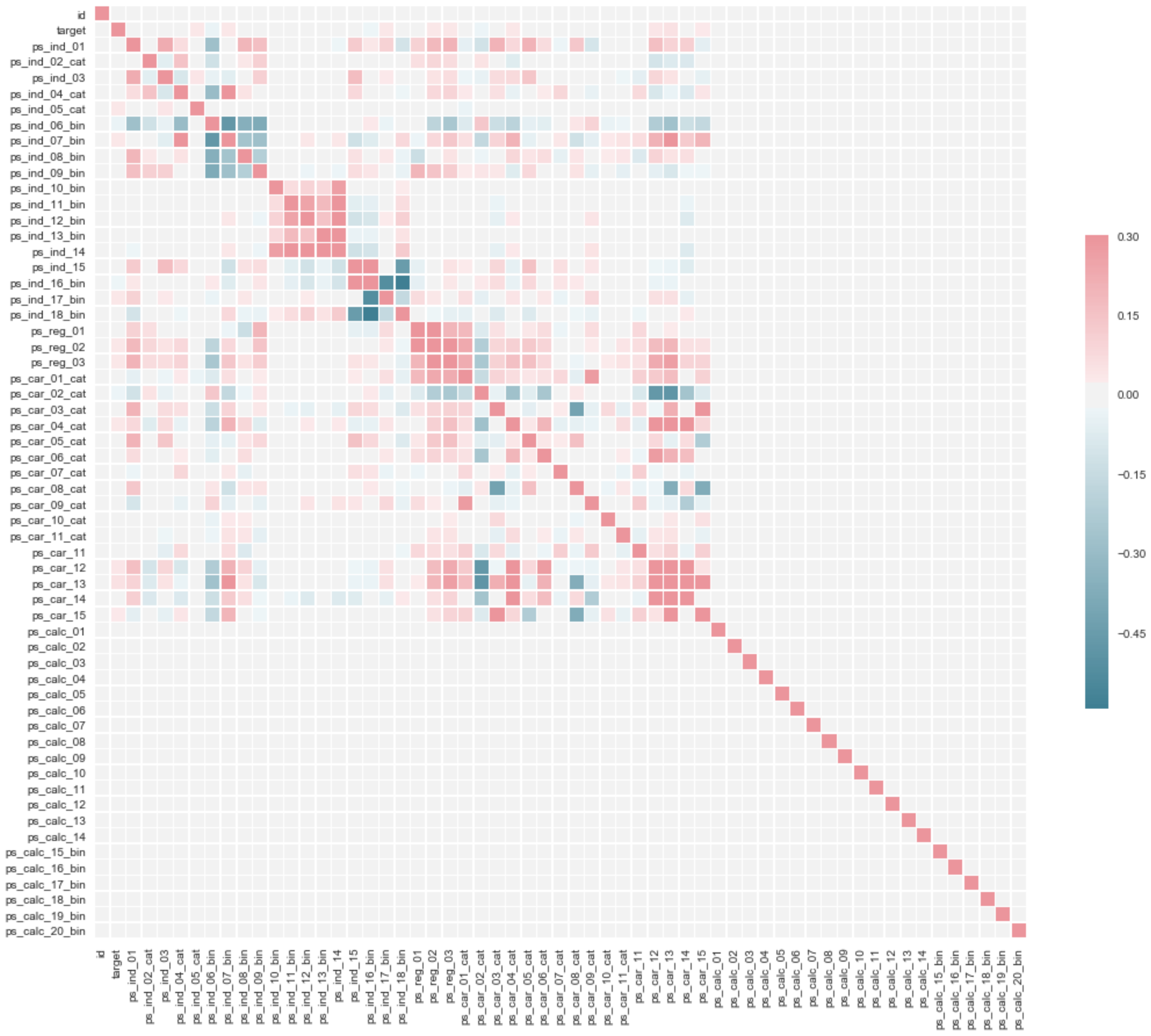


Figure 5.1 Correlation heatmap

Every ‘ps_calc_’ feature has no significant correlation with other features. Thus, they are dropped to initially reduce the dimensionality of the feature space. To make it more convincing, figure 5.2 shows a simple feature score.

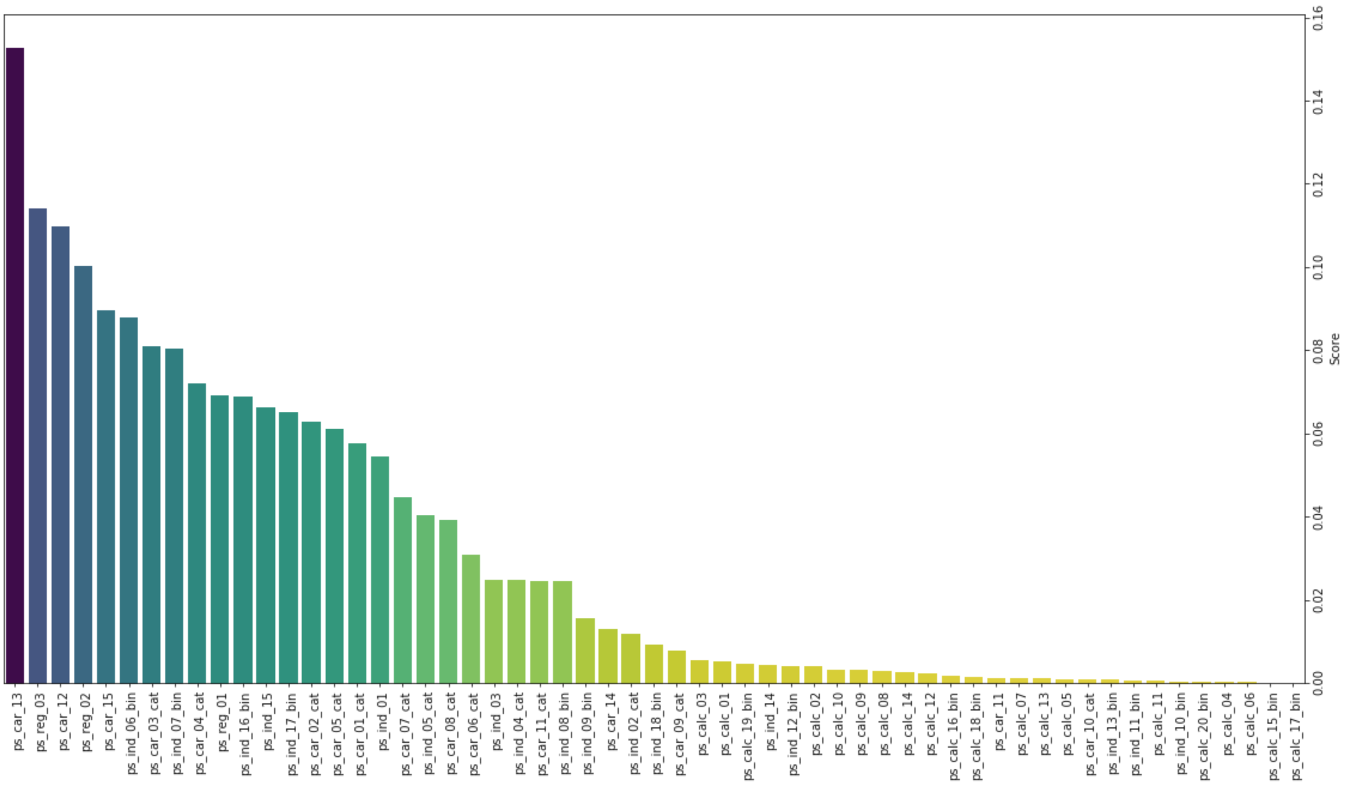


Figure 5.2 Feature score [6]

All ‘ps_calc_’ features have low rank in the scorer, and this further indicates that they can be removed.

5.2 Feature engineering

To make our model more powerful, new predictive features are built.

5.2.1 Interaction term

The first one is *ps_car_13 * ps_reg_03*. According to figure 5.2 and kernels on Kaggle, these two features are always the top 2 important features; consequently, it is intuitive to add an interaction term of them to create a possible predictive variable.

5.2.2 Number of NA’s per id

Another new created feature is the number of missing values per id. This feature indicates how much unknown information a driver has. Figure 5.3 shows claim rates versus the number of missing values per id.

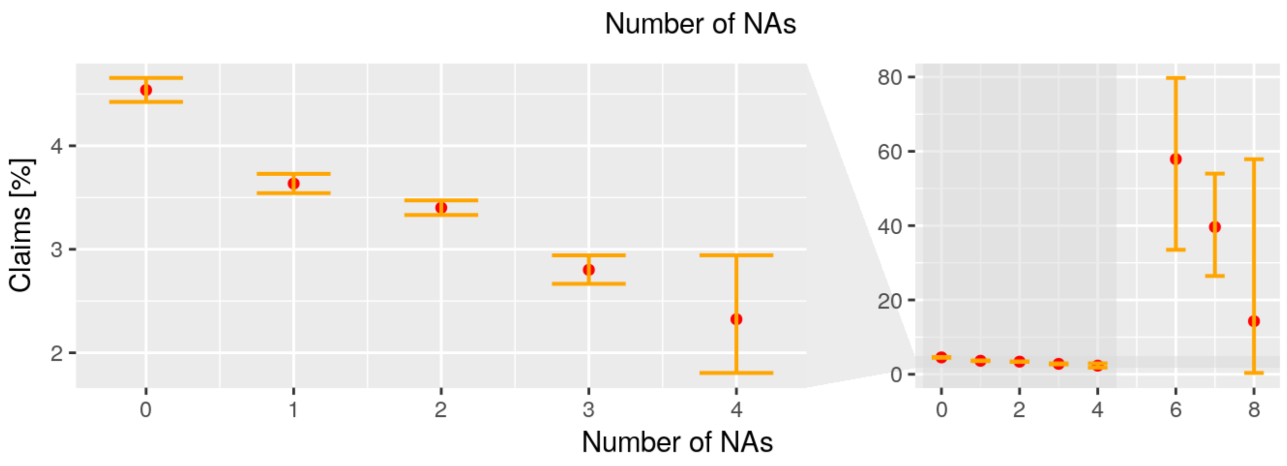


Figure 5.3 Claim rates vs. number of NaN's

As the number of missing values per id goes up from 0 to 4, the claim rate slightly drops from 5% to around 2%, and there is no data sample having 5 missing values. Almost 60% of data having 6 missing values are claimed, while 40% of data having 7 missing values are claimed, with a decreasing trend as well. 8 is not statistically significant due to the limited number of samples [7]. These patterns indicate that this new feature might contribute to the model.

5.2.3 Sum of binary features

The sum of 'ps_ind_**_bin' per id, and the sum of 'ps_calc_**_bin' per id are also added into the model. Figure 5.4 presents claim rates versus these two variables.

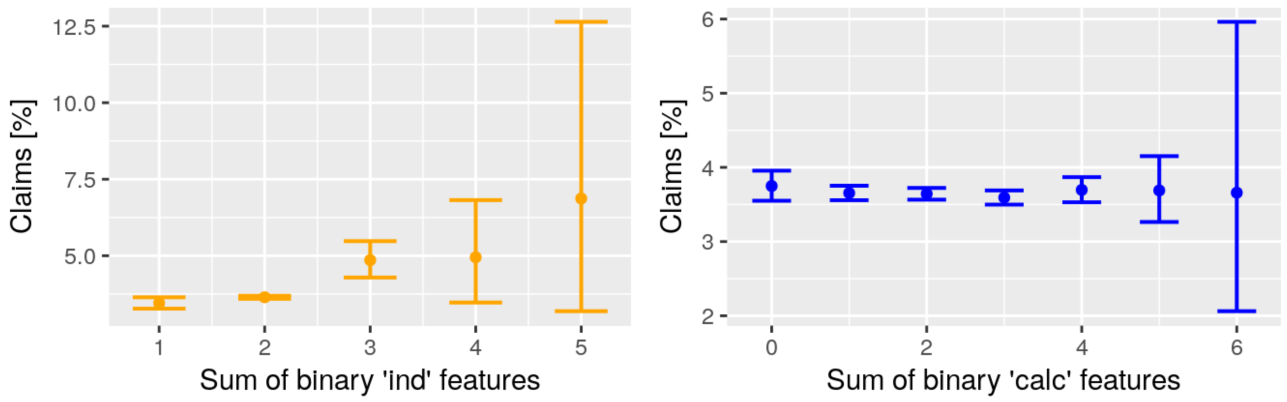


Figure 5.4 Claim rates vs. sum of binary 'ind' features and sum of binary 'calc' features

The claim rate increases as the sum of binary 'ind' features per id increases, although 4 and 5 are not statistically significant. This pattern makes the new feature possibly helpful in the model. On the other hand, the claim rate for the sum of binary 'calc' features remains stable, which further demonstrates that 'ps_calc_' features are not predictive and can be dropped [7].

5.2.4 Difference measure for binary features

Two more features are created, i.e. two difference measures for binary 'ind' features and binary 'calc' features. The reference row is the median for each, and then the reference row is subtracted from binary values for each row of the data, with the sum of the absolute differences being a new feature. Figure 5.5 shows claim rates versus absolute difference of binary ‘ind’ features and absolute difference of binary ‘calc’ features.

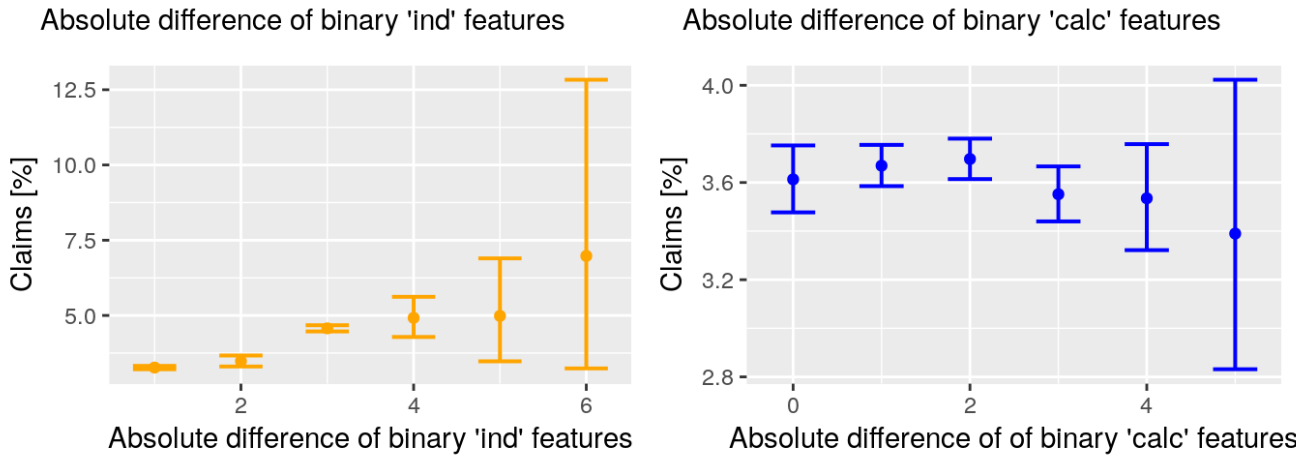


Figure 5.5 Claim rates vs. absolute difference of binary ‘ind’ features and binary ‘calc’ features

The claim rate goes up as the binary ‘ind’ feature deviates from the reference, although 5 and 6 are not statistically significant. This pattern makes this new feature possibly predictive. For the absolute difference of binary ‘calc’ features, there is a slight increase and then a decline in claim rate, although 4 and 5 are not statistically significant. However, this pattern is not very obvious because of the uncertainty of confidence level, which further shows that ‘ps_calc_’ features are not predictive.

5.2.5 Conclusion

Data preparation, data transformation and feature engineering are very important before modeling. It turns out that the above created features provide high importance to our model. This will be presented in 5.4. To increase final performance, more feature engineering should be considered.

5.3 Pre-processing and feature extraction

This section presents the methods for filling missing data. Also, three methods are discussed for feature extraction: Principle Component Analysis (PCA), Non-negative Matrix Factorization (NMF) and Independent Component Analysis (ICA).

5.3.1 Missing data information

There are 13 features containing missing data in training set and 12 features containing missing data in test set. Table 5.2 and table 5.3 list these missing features, with the ratio of missing data in training set and test set respectively.

Missing value feature	Ratio of missing data
ps_ind_02_cat	3.6290e-4
ps_ind_04_cat	1.3945e-4
ps_ind_05_cat	0.0098
ps_reg_03	0.1811
ps_car_01_cat	1.7977e-4
ps_car_02_cat	8.4004e-06
ps_car_03_cat	0.6909
ps_car_05_cat	0.4478
ps_car_07_cat	0.0193
ps_car_09_cat	9.5596e-4
ps_car_11_cat	8.4004e-06
ps_car_12_cat	1.6801e-06
ps_car_14_cat	0.0716

Table 5.2 Ratio of missing data for each feature in training set

Missing value feature	Ratio of missing data
ps_ind_02_cat	]3.4386e-4
ps_ind_04_cat	1.6241e-4
ps_ind_05_cat	0.0098
ps_reg_03	0.1811
ps_car_01_cat	1.7921e-4
ps_car_02_cat	5.6003e-06
ps_car_03_cat	0.6910
ps_car_05_cat	0.4484
ps_car_07_cat	0.0194
ps_car_09_cat	9.8229e-4
ps_car_11_cat	1.1201e-06
ps_car_14_cat	0.0715

Table 5.3 Ratio of missing data for each feature in test set

Figure 5.6 provides equivalent visualization that can help deal with missing values.

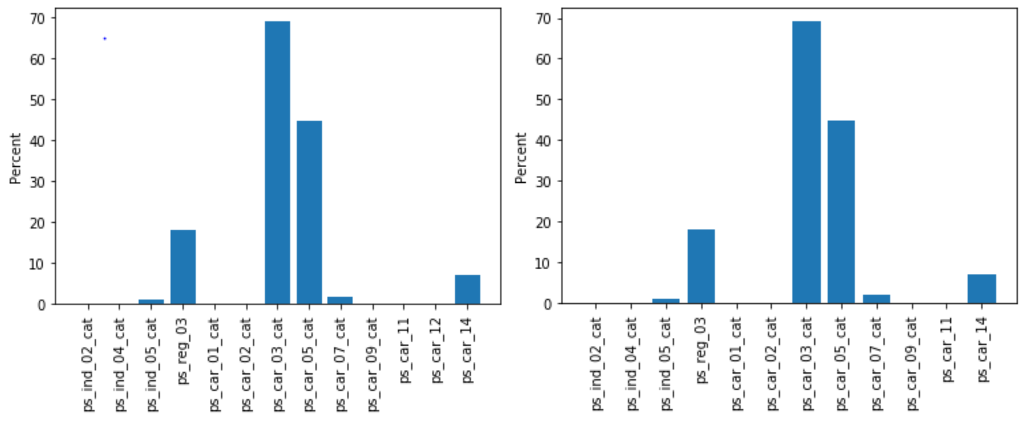


Figure 5.6 Ratio of missing data for each feature in training and test data

Due to a huge number of training set, filling data inside cross-validation loops by fitting algorithms will cost much computation time and is impractical. Therefore, missing data is imputed before cross-validation. Labels are ignored when filling both training and test data, rather than filling the training set with the same output label, in case of cheating in CV loops.

One may argue that missing values can still be filled with less computation time, inside CV loops by simple methods such as mean, median or mode imputation. The trade-off for this large dataset is whether performing imputation in CV loops using computationally cheap methods, or performing imputation out of CV loops using precise methods. For this dataset, former is preferred, as figure 5.6 indicates that there are many missing values, and the point is that mean, median or mode imputation introduce larger amount of noise than fitting algorithms.

5.3.2 Imputation

Before imputation, one-hot encoding is performed on categorical features. For example, if a categorical feature has 4 categories, these 4 categories are interpreted as 1: 0001, 2: 0010, 3: 0100, and 4: 1000, which means that the feature is expanded to 4 binary features.

Missing data is imputed with different methods. Because *ps_car_03_cat* and *ps_car_05_cat* have around 70% and 45% of missing values respectively, both can be removed. *ps_reg_03* contains about 18% of missing data. Imputation using fitting algorithms takes long time. Consequently, K-nearest-neighbor (KNN) method is applied first to find the nearest 100 points and then the missing value is filled by averaging the 100 points. Fitting algorithms are not applied here.

For *ps_ind_02_cat, ps_ind_04_cat, ps_ind_05_cat, ps_car_01_cat, ps_car_02_cat, ps_car_07_cat, ps_car_09_cat,* the highest missing ratio is around 8%, thus fitting algorithms can be applied now. K = 2000 (K is chosen to be 2000 since there are around 200 features used for predicting missing value. The rule of thumb is the number of examples should be at least 10 times of the dimensionality, to prevent overfitting) nearby points are found by KNN, and random forest classifier is applied to fit these nearby points. To build a random forest, the parameters are set as below: 120 for 'randomFeatures', 0.8 for 'bagSize', and 2000 for the number of decisions trees.

Similarly for *ps_car_11, ps_car_12 and ps_car_14,* 2000 nearby points are found first, and random forest regressor is applied to predict the missing value.

5.3.3 Principle Component Analysis

PCA is a technique used to maximize variation and bring out string patterns in dataset, especially for high-dimensional feature space. PCA tries to find certain important components which can separate target apparently [8]. Figure 5.7 shows an example in 2-dimension. The goal is to reduce dimensionality down to 1.

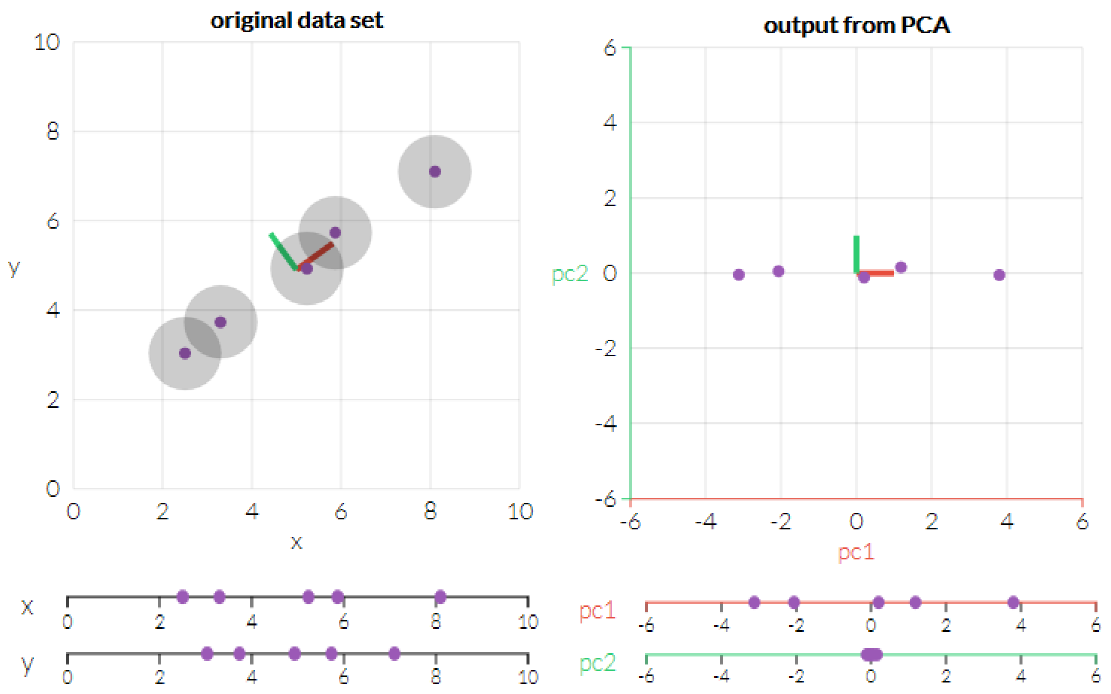


Figure 5.7 Example for PCA in 2-dimension

Suppose the original data is projected onto the direction of x, large amount of information on y direction will be lost. However, PCA finds the direction for projection that maximizes variance, i.e. the red component pc1 in figure 5.7, while the component in green pc2 contains the information that will be lost once projected. Clearly, only a small amount of information on is lost, while most of the data information is kept by PCA.

Steps for using PCA: PCA subtracts mean from each column for matrix **A**, and then computes the covariance matrix. Next, PCA computes eigenvectors and eigenvalues of the covariance matrix. The K eigenvectors corresponding to the first K largest eigenvalues form a projection matrix **P**, which are the K components containing most information. Finally, the original design matrix can be projected on a K-dimensional space by multiplying matrix **A** by **P**.

5.3.4 Independent Component Analysis

Another method for dimensionality reduction is Independent Component Analysis (ICA). ICA is similar to PCA, except that ICA can be applied to distribution models other than Gaussian distribution. ICA mainly discriminates independent features from mixed signals and maximize them [9].

According to Central Limit Theorem, the sum of independent random variables will converge to Gaussian distribution, which means that independence will become poor with higher Gaussianity. For ICA model,

$$x = As, \quad s = A^{-1}x$$

Let

$$y=\omega^T x,\quad z=A^T\omega.$$

Then

$$y=\omega^T x=\omega^T As=z^T s.$$

Therefore, y is a linear combination of s, and y should have higher Gaussianity than s. When

$$\omega^T\approx A^{-1},$$

then

$$y=\omega^T x=A^{-1}x=s.$$

y will have the smallest Gaussianity when y = s. Now the problem is to find the optimal ω that minimizes the Gaussianity of $\omega^T x$. EM algorithm, gradient descent and FastICA algorithm can be applied to find the best solution.

5.3.5 Non-negative Matrix Factorization

Sometimes data is nonnegative by nature. PCA may give negative outcomes, while Non-negative Matrix Factorization (NMF) can avoid that. NMF decomposes original matrix **V** into two non-negative matrices **W** and **H**, which leads to sparsity and reduces feature space [10].

Original dataset **V** is F * N, where F is dimensionality and N is the number of samples. **W** is a F * K dictionary matrix which can be considered as a feature space. **H** is a K * N expansion or activation matrix.

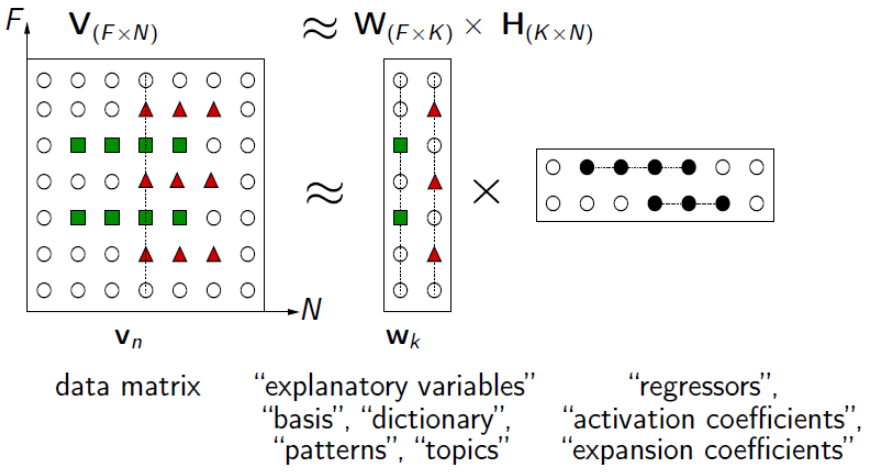

F
$\mathbf{V}_{(F\times N)}$
$\approx$ $\mathbf{W}_{(F\times K)}$ $\times$ $\mathbf{H}_{(K\times N)}$
N
$\mathbf{v}_n$
$\mathbf{w}_k$
data matrix
"explanatory variables" "basis", "dictionary", "patterns", "topics"
"regressors", "activation coefficients", "expansion coefficients"

Figure 5.8 General formulation of NMF

Figure 5.8 shows general formulation of NMF. The point is to choose K. A larger value of K leads to a better solution, nonetheless increasing the complexity of this model and slowing down convergence. As a result, a proper K makes a difference to feature extraction.

After NMF algorithm, two matrices **W** and **H** are derived from the original matrix **V**. **W** * **H** gives a sparser matrix **V'** as the new input for training model. Though **V'** has the same dimension as **V**, it has more zeros entries than **V**, which can be considered as a reduced feature space.

5.4 Modeling

After modeling preparation, the model and classifiers can be trained. For both XGBoost classifier and logistic regression, 595213 training samples are fitted in a 203-dimensional feature space. The difference is that logistic regression takes advantage of feature extraction to reduce dimensionality, while XGBoost chooses the best dimensionality by itself. The modeling process utilizes 5-fold cross-validation to choose the optimal parameters. These parameters include l1 and l2 regularization terms which prevent overfitting if they are tuned well.

5.4.1 Evaluation metric

For this dataset, normalized Gini coefficient is used to evaluate the performance. This is because true probability values are not auto insurance companies care about; instead, they look at the relative order of predicted probabilities [7]. If actual labels are sorted in ascending order of predicted probabilities, a model is perfect if the actual labels have all 0's on the left and all 1's on the right. In this case, normalized Gini coefficient is a good metric.

Normalized Gini coefficient can be calculated either by its principle or by AUC in the form of equation (5.1)

$$\text{Normalized Gini coefficient} = 2 * \text{AUC} - 1 \tag{5.1}$$

Since it is proportional to AUC, the model can be trained using metric AUC.

5.4.2 Training XGBoost classifier

Since XGBoost has 10 parameters to tune, it is impractical to perform grid search on all combinations of them. For example, if each parameter has only 3 options, there will be 3^10 combinations to search. Fortunately, some parameters can be tuned independently, without influencing the performance. The general rules are fix learning rate to obtain the best number of trees at first, and afterwards tune tree-specific parameters (max_depth, min_child_weight, gamma, subsample, etc) for decided learning rate and number of trees [11]. Then the regularization parameters are tuned, followed by reducing learning rate and increasing the best number of trees for updated parameters.

5.4.2.1 Tuning number of trees

Learning rate is fixed to 0.1, a relatively high value, to make convergence fast. Table 5.4 presents the start values for other parameters.

Parameters	Value	Reason
objective	binary:logistic	This is a binary classification problem
eval_metric	auc	The metric is Gini coefficient
max_depth	5	Start values in [4, 5, 6] typically works
min_child_weight	1	Default, common choice
gamma	0	Default, common choice
subsample	0.8	Default, common choice
colsample_bytree	0.8	Default, common choice
scale_pos_weight	1	Default, common choice
alpha	0	Default, common choice
lambda	1	Default, common choice

Table 5.4 Start values for parameters

Then the optimal number of trees is obtained by 5-fold cross-validation. This gives 113.

5.4.2.2 Tuning max_depth and min_child_weight

Because max_depth and min_child_weight have the largest influence on XGBoost classifier, they are the next parameter set to tune. A grid search is performed on a wide range, {'max_depth': [3, 5, 7, 9], 'min_child_weight': [1, 3, 5, 7, 9]}. Table 5.5 shows the average AUC's for different pairs of these two parameter values.

	1	3	5	7	9
3	0.63718	0.63732	0.63701	0.63711	0.63708
5	0.63932	0.63977	0.63967	0.64041	0.63977
7	0.63686	0.63743	0.63730	0.63814	0.63802
9	0.63706	0.63107	0.63321	0.63399	0.63458

Table 5.5 Average AUC

The highest AUC occurs when max_depth = 5 and min_child_weight = 7. Thus, a grid search with deep step size is performed around max_depth = 5 and min_child_weight = 7 ({'max_depth': [4, 5, 6], 'min_child_weight': [6, 7, 8]}). Table 5.6 lists the results.

	6	7	8
4	0.63937	0.63921	0.63945
5	0.63994	0.64041	0.64018
6	0.63935	0.64002	0.63957

Table 5.6 Average AUC with deep step size

Obviously, the optimal max_depth is 5, and the optimal min_child_weight is 7. Currently the highest cross-validation AUC is 0.64041.

5.4.2.3 Tuning gamma

The next parameter is gamma. Its values are picked from 0 to 2 using logspace(). Figure 5.9 shows average AUC versus gamma. The optimal gamma is 0, and currently the highest AUC is still 0.64041.

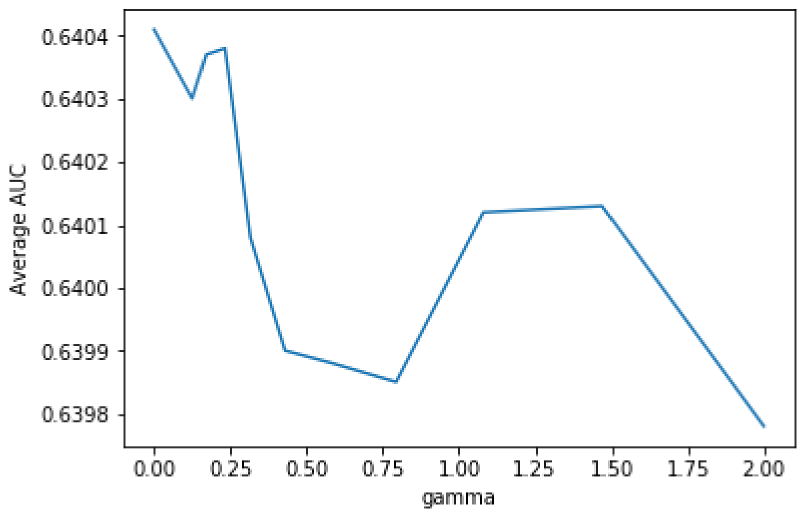


Figure 5.9 Average AUC vs. gamma

5.4.2.4 Updating number of trees

A good idea is to re-compute the optimal number of trees, using the previous updated parameters. The cross-validation gives 152.

5.4.2.5 Tuning subsample and colsample_bytree

A grid search for subsample and col_sample_bytree is performed on a wide range, {'subsample': [0.5, 0.6, 0.7, 0.8, 0.9], 'col_sample_bytree': [0.5, 0.6, 0.7, 0.8, 0.9]}. Table 5.7 illustrates the average AUC for pairs of these two parameter values.

subsample col_sample_bytree	0.5	0.6	0.7	0.8	0.9
0.5	0.63791	0.63897	0.63893	0.63982	0.64004
0.6	0.63773	0.63950	0.63946	0.63919	0.63919
0.7	0.63823	0.63971	0.63934	0.64041	0.64035
0.8	0.63786	0.63895	0.64004	0.64047	0.64008
0.9	0.63929	0.63949	0.63947	0.64022	0.64013

Table 5.7 Average AUC

The highest average AUC occurs when col_sample_bytree = 0.8 and subsample = 0.8; thus a grid search with deep step size is performed around 0.8 ({'subsample': [0.75, 0.8, 0.85], 'colsample_bytree': [0.75, 0.8, 0.85]}). Table 5.8 lists the results.

subsample col_sample_bytree	0.75	0.8	0.85
0.75	0.63967	0.64031	0.64051
0.8	0.63955	0.64047	0.63986
0.85	0.64046	0.63976	0.64026

Table 5.8 Average AUC with deep step size

Obviously, the optimal col_sample_bytree is 0.75 and the optimal subsample is 0.85. Currently the highest cross-validation AUC increases to 0.64051.

5.4.2.6 Tuning alpha and lambda

Now regularization terms are tuned to prevent overfitting. Both alpha and lambda start with a very wide range ({‘alpha’: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], ‘lambda’: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}). The optimal values are alpha = 10 and lambda = 3. Figure 5.10 shows the average AUC vs. lambda when alpha = 10.

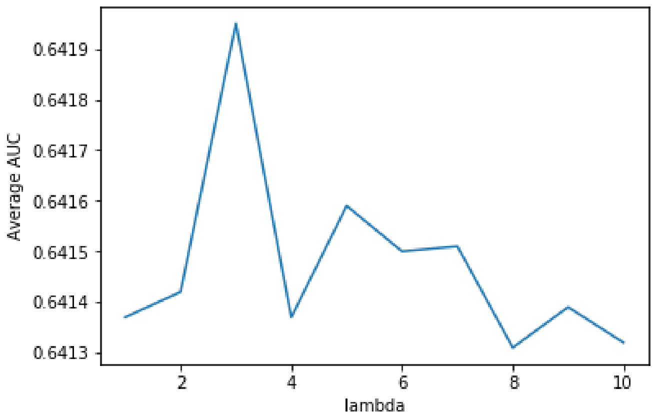


Figure 5.10 Average AUC vs. lambda, when alpha = 10

Although the average AUC reaches to the highest, i.e. 0.64195, this number is not very reliable because it is within twice and triple times of standard deviation. It also peaks excessively high, which deviates the overall declining trend in AUC. Therefore, alpha = 10 and lambda = 3 is only a candidate for optimal parameter.

Then a grid search with different range is performed ({'alpha': [10, 11, 12], 'lambda': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}). Alpha = 12 and lambda = 3 gives the second highest average AUC. Figure 5.11 shows the average AUC vs. lambda when alpha = 12.

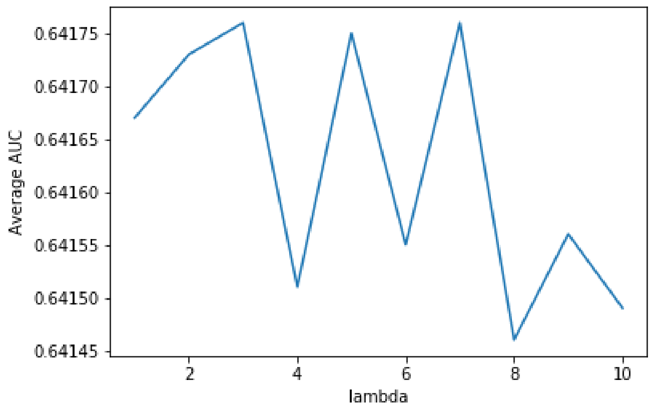


Figure 5.11 Average AUC vs. lambda, when alpha = 12

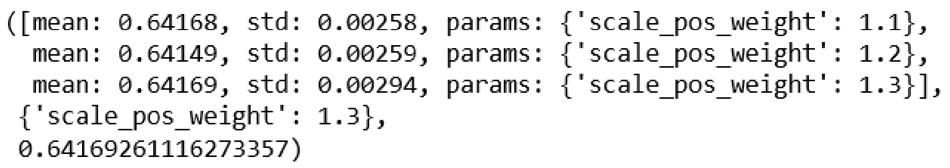
The overall trend is also decreasing; however, it is more reliable to trust alpha = 12, lambda = 3 than alpha = 10, lambda = 3, since all values in figure 5.11 are within a reasonable range. It can also be proved that all values fall into 1.5 times of standard deviation. Now, the candidate for optimal parameters becomes alpha = 12 and lambda = 3.

A further grid search with alpha = 13 and 14 is conducted. The AUC's are lower than our candidate. At last, a grid search with deep step size is completed. The values for alpha are generated by logspace() in interval [11, 13], and the values for lambda are generated by logspace() in interval [2, 4]. The final optimal parameters are alpha = 12.1789 and lambda = 3.6.

After applying l1 and l2 regularization, the highest cross-validation AUC increases significantly to 0.64198, which means that overfitting has been reduced. This can be further explained by the large alpha that reduces the model complexity by generating sparse solutions in our 203-dimensional feature space.

5.4.2.7 Tuning scale_pos_weight

The dataset is imbalanced; consequently, it makes sense to try larger scale_pos_weight. According to discussion and kernels on Kaggle, this hyperparameter influences performance negatively on this dataset. Therefore, only a few values are tried here. Figure 5.12 shows the average AUC vs. scale_pos_weight.



```
([mean: 0.64168, std: 0.00258, params: {'scale_pos_weight': 1.1},
  mean: 0.64149, std: 0.00259, params: {'scale_pos_weight': 1.2},
  mean: 0.64169, std: 0.00294, params: {'scale_pos_weight': 1.3}],
 {'scale_pos_weight': 1.3},
 0.64169261116273357)
```

Figure 5.12 Average AUC vs. scale_pos_weight

AUC decreases from our highest 0.64198 to 0.64149, if more weight is imposed on the positive class. The reason is that the weight tends to classify some 0 to 1, thus changing the relative order of predicted probabilities. This changed relative order will lower AUC and Gini coefficient. If the problem is credit risk prediction, then scale_pos_weight is more useful; however, this parameter seems not to be appropriate for this dataset.

5.4.2.8 Reducing learning rate and increasing number of trees

The last step for tuning parameters for XGBoost is to reduce learning rate and add more trees. Table 5.9 presents the optimal number of trees with average AUC corresponding to different learning rates.

Learning rate	Number of trees	Training AUC	Validation AUC
0.08	198	0.678578	0.642732
0.07	229	0.679101	0.642578
0.06	333	0.686358	0.642653
0.05	389	0.685247	0.642901
0.04	440	0.682202	0.643206
0.03	669	0.686796	0.643319
0.02	996	0.686514	0.643455
0.01	1851	0.684132	0.643525

Table 5.9 Optimal number of trees and average AUC corresponding to learning rate

As learning rate goes down, the cross-validation AUC keeps increasing. Therefore, the final learning rate is 0.01 with the optimal number of trees = 1851. The highest CV AUC increases to 0.643525.

5.4.2.9 Cross-validation performance using optimal parameters

Now the XGBoost classifier is trained inside 5-fold cross-validation loop to observe overfitting, using the optimal parameters. Table 5.10 illustrates the Gini coefficient on training and validation set at each loop, as well as the average Gini coefficient on entire training set.

	1	2	3	4	5	Average	Std
CV training set	0.366086	0.370268	0.369022	0.368144	0.368343	0.368373	0.001364
CV validation set	0.298554	0.280624	0.287609	0.283527	0.287713	0.287713	0.006087
Entire training set	0.287529726333						

Table 5.10 Gini coefficient on CV training set, CV validation set, and entire training set

Although the Gini coefficient on CV training set is higher than on CV validation set, it shows no serious overfitting, as the Gini coefficient being converted to AUC. This can be explained by table 5.9.

The average Gini coefficients on validation sets and on entire training set are not very low compared with public leaderboard; thus, a final model can be trained with the optimal parameters using the entire training set.

5.4.2.10 Feature importance

The final model gives the feature importance. 186 Features are split when growing trees, 17 features are not used at all, and most 'ps_car_' features are only picked a few times. Figure 5.13 shows the top 20 features.

In terms of created features in 5.2, the interaction term *ps_car_13 * ps_reg_03*, difference measure for binary 'calc' features, number of binary 'calc' per id, number of missing values per id, and difference measure for binary 'ind' features show large predictive power. They are circled in figure 5.13. The number of binary 'ind' per id also ranks 64th among 203. It turns out our feature engineering works.

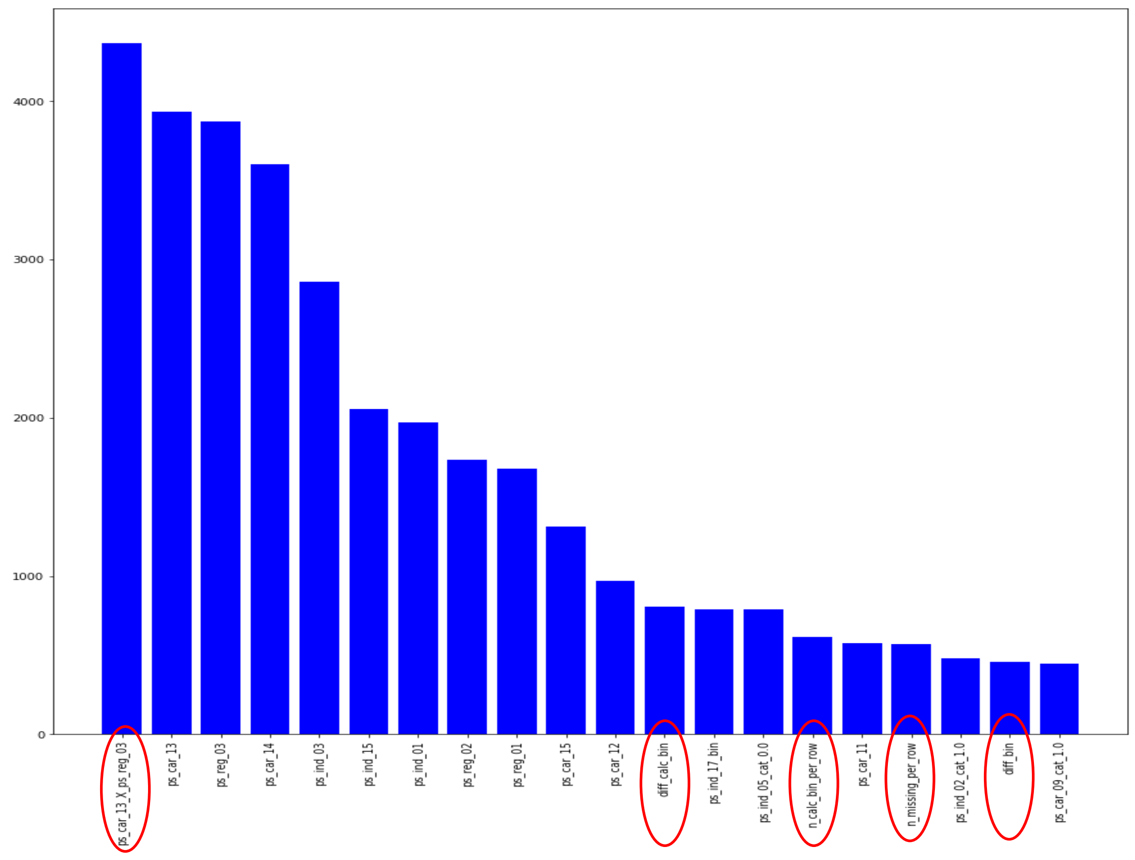


Figure 5.12 Feature importance

5.4.3 Training logistic regression model

In this section, feature extraction is brought into picture. The parameters for logistic regression are alpha (L1 regularization term) and lambda (L2 regularization term). In this report, they are both included into the model. This is also called Elastic net regularization.

5.4.3.1 Feature extraction

Before tuning parameters, the three feature extraction methods are executed in comparison with performance without feature extraction. It turns out that all of them lowers down Gini coefficient.

Taking PCA as an example, table 5.11 and figure 5.13 present the proportional relationship between Gini coefficient and number of features after PCA. First 170 components contain more than 97% of information of all data, which can fully represent the original dataset.

Number of features	Gini on training set	Gini on validation set
160	0.2579	0.2560
170	0.2610	0.2584
180	0.2614	0.2587
190	0.2614	0.2587

Table 5.11 Gini coefficient vs. number of features

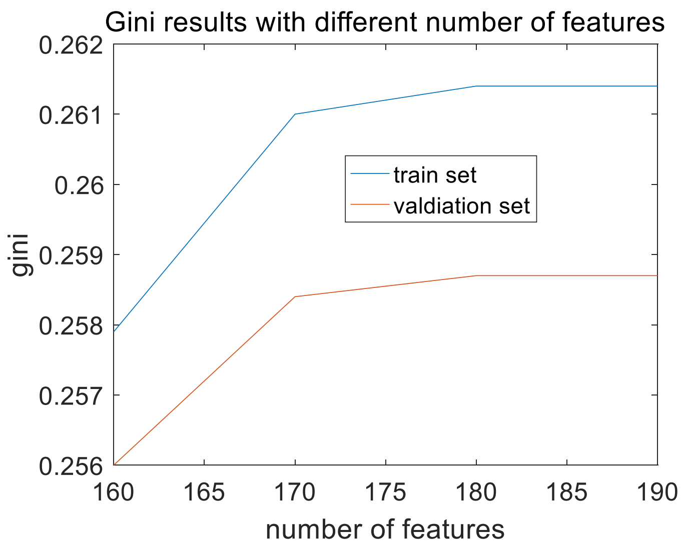


Figure 5.13 Gini coefficient vs. number of features

Performance increases significantly as the number of features grows at first, and then keeps going up slowly. For this reason, the three feature extraction methods would ruin the performance, although they save time for convergence.

5.4.3.2 Ensemble method

Since the dataset is very imbalanced, logistic regression requires long time to converge, or may even not converge. To improve the model, ensemble method is used, i.e. many under-sampled and 1:1 balanced subsets are trained and averaged. Figure 5.14 shows performance vs. number of subsets.

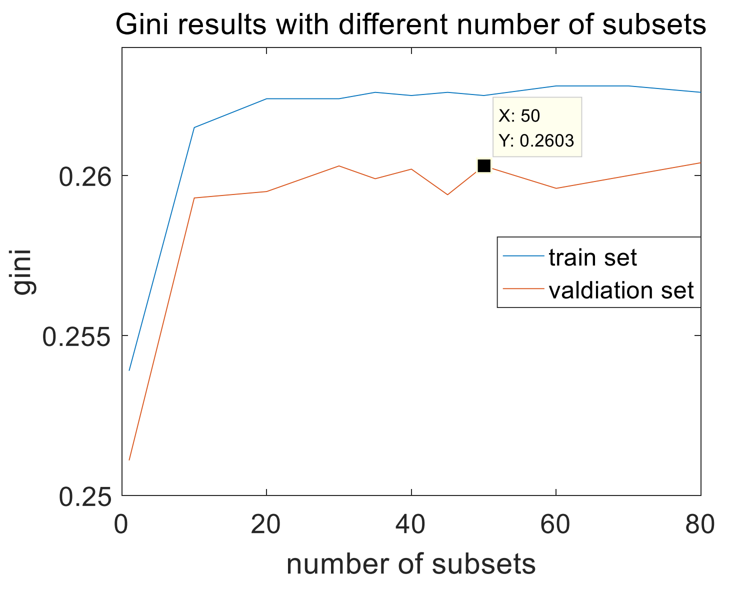


Figure 5.14 Gini coefficient vs. number of subsets

Generally, a larger number of subsets leads to better results, while consuming more time. To make a balance between efficiency and performance, 50 is chosen. This number also makes sense because as the number of subsets becomes larger than 50, performance only increases marginally.

5.4.3.3 Tuning alpha and lambda

L1 and L2 regularization terms are tuned now. L1reults in sparse solutions, which is useful for dealing with high dimensionality, whereas L2 penalizes complexity.

A grid search is performed on a wide range, {‘alpha’: [0, 2, 4, …,18, 20], ‘lambda’: [0, 2, 4, …, 20]}. The largest Gini coefficient occurs when alpha = 10, lambda = 18. After that, another grid search is conducted on a deeper range, {‘alpha’: [8 9 10 11 12], ‘lambda’: [16 17 18 19 20]}. This gives alpha around 11 and 12, lambda around 18. To obtain more accurate values, a deeper step size is applied using logspace() function. Table 5.12 lists the Gini results after the third round.

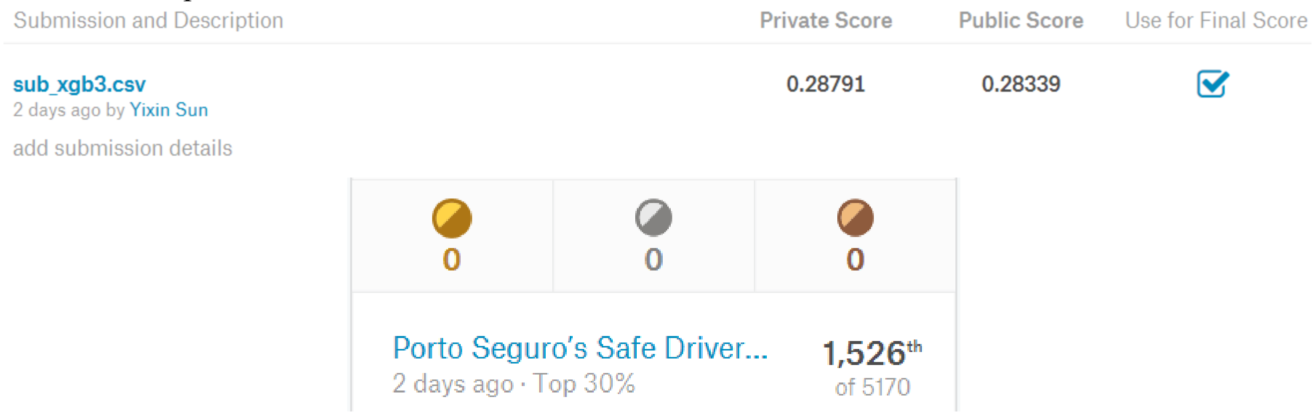
alpha lambda	11.0000	11.2419	11.4891	11.7417	12.0000
17.0000	0.261535386	0.261559259	0.261583828	0.261609705	0.261633786
17.2722	0.261535349	0.261559896	0.261583780	0.261609551	0.261633943
17.5489	0.261535528	0.261559567	0.261583680	0.261609491	0.261633760
17.8299	0.261536677	0.261559534	0.261583626	0.261609440	0.261633953
18.1155	0.261536656	0.261559632	0.261583636	0.261609100	0.261633940
18.4056	0.261536584	0.261569906	0.261583687	0.261609129	0.261633891
18.7005	0.261536749	0.261559778	0.261583641	0.261609461	0.261634133
19.0000	0.261536921	0.261559628	0.261583916	0.261609697	0.261633500

Table 5.11 Gini coefficients with deep range of alpha and lambda

The maximum Gini coefficient is 0.261634133, the optimal alpha is 12, and the optimal lambda is 18.7005. With selected parameters, our model is finally trained on the full training set and is evaluated on the test set.

6. Final results

Logistic regression has the final Gini coefficients 0.27539412 on training set and 0.25934249 on test set. Because XGBoost classifier is more robust and powerful for this dataset than logistic regression, the best submission is predicted by the final XGBoost model. Figure 6.1 shows the best submission result and the place in leaderboard.



Submission and Description	Private Score	Public Score	Use for Final Score
sub_xgb3.csv 2 days ago by Yixin Sun add submission details	0.28791	0.28339	✓

●	●	●
0	0	0

Porto Seguro’s Safe Driver...
2 days ago · Top 30%

1,526th
of 5170

Figure 6.1 Best submission and place in leaderboard

For a single XGBoost model, this performance makes sense. From all kernels and discussion on Kaggle, the highest performance that XGBoost can achieve is only 0.284 on public leaderboard score. Our 0.28339 is not very far away. To climb to a higher place on leaderboard, obviously a single model is not enough, and an ensemble or stacking of several models should be applied.

7. Interpretation

Gradient boosting is a good fit for this problem domain because of high dimensionality, large number of missing values, non-clear distinguish between 2 classes, etc., although parameters have to be tuned carefully.

However, logistic regression model does not perform well on such a dataset. There are several reasons. First, Logistic regression is sensitive to missing values. Original dataset contains a large number of missing values. Some of them are important features. Imputation still introduces noises. As comparison, logistic regression is not as appropriate as XGBoost for this dataset.

Second, data preprocessing is very important for logistic regression. More data preparation, data cleaning, feature engineering, feature transformation, and dimensionality reduction should be considered, each of which will have a positive impact on logistic regression. Therefore, limited feature engineering leads to a limited performance in this report.

Third, logistic regression works well on linear problems. This dataset has small linearity with limited data preprocessing, making logistic regression hard to separate 0, 1 classes well in the feature space.

Forth, figure 5.1 shows that many features have high correlation with others. This is called multicollinearity, which can result in unstable coefficients for logistic regression. XGBoost is robust to multicollinearity, which further indicates that it is more suitable than logistic regression here.

In conclusion, for complex problems, logistic regression requires a large amount of data preparation, feature engineering, feature transformation, etc. to achieve high performance. The preparation work has not been explored in depth in this report.

8. Contributions of each team member

For modeling preparation, Yixin Sun contributes to feature engineering. Ben Zhang contributes to feature extraction. Both of them are responsible for missing data imputation.

For modeling, Yixin Sun contributes to training XGBoost model, while Ben Zhang contributes to training logistic regression model. Both of them are responsible for ensemble method (i.e. averaging many under-sampled training data) to improve performance.

9. Summary and conclusions

Data preparation, data cleaning, feature engineering, feature transformation, etc. are important for all machine learning problems, which usually cost 80% of time. XGBoost is more robust to missing values, high dimensionality and non-linearity, while logistic regression is simpler to train and easier to interpret, although data preprocessing should be treated more carefully.

Future work includes a weighted or unweighted ensemble or stacking of robust models, such as neural networks, and lightgbm. This makes prediction more accurate and can lead to a more stable relative order of predicted probabilities.

Reference

[1] Kaggle, Inc. Porto Seguro’s Safe Driver Prediction[DB/OL]. https://www.kaggle.com/c/porto-seguro-safe-driver-prediction.

[2] Kevin, P·Murphy. Machine Learning: A Probabilistic Perspective[M]. Cambridge, US:The MIT Press, 2012.

[3] DMLC. XGBoost Parameters[EB/OL]. http://xgboost.readthedocs.io/en/latest/parameter.html#xgboost-parameters.

[4] Jeff, Howbert. Machine Learning Logistic Regression [EB/OL]. http://courses.washington.edu/css490/2012.Winter/lecture_slides/05b_logistic_regression.pdf. 2012.

[5] Jason, Browlee. How To Implement Logistic Regression With Stochastic Gradient Descent From Scratch With Python[EB/OL]. https://machinelearningmastery.com/implement-logistic-regression-stochastic-gradient-descent-scratch-python/. 2016.

[6] Olivier. Simple Feature Scorer[EB/OL]. https://www.kaggle.com/ogrellier/simple-feature-scorer. 2017

[7] Heads or Tails. Steering Wheel of Fortune - Porto Seguro EDA[EB/OL]. https://www.kaggle.com/headsortails/steering-wheel-of-fortune-porto-seguro-eda. 2017

[8] Victor, Powell. Principal Component Analysis[EB/OL] http://setosa.io/ev/principal-component-analysis/. 2014

[9] Juan, Qi. Indepedent Component Analysis[EB/OL]. https://wenku.baidu.com/view/b9598f060740be1e650e9a58.html. 2017

[10] Slim, ESSID, &, Alexey, OZEROV. A TUTORIAL ON NONNEGATIVE MATRIX FACTORISATION WITH APPLICATIONS TO AUDIOVISUAL CONTENT ANALYSIS[EB/OL]. https://perso.telecom-paristech.fr/~essid/resources.htm. 2014

[11] Aarshay, Jain. Complete Guide to Parameter Tuning in XGBoost (with codes in Python)[EB/OL]. https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/. 2016