Building Claim Prediction - GENERALI

Alexandre TRENDEL, Nicolas HUBERT, Youssef ALEIAN

Mars 2019

1 Introduction

1.1 L'entreprise, Generali

Generali est une compagnie d'assurance italienne, classée troisième plus importante entreprise au monde dans son secteur.

Elle propose notamment des assurances habitations, qui permettent une couverture en cas de sinistre. Dans le cadre de ce challenge, nous nous intéresserons à ce type d'assurance.

1.2 Le challenge

L'objectif de ce challenge est de prédire si un bâtiment assuré par GENERALI est susceptible de déposer *a minima* une réclamation d'assurance durant une période donnée. Il s'agit donc d'un problème de classification supervisée – les valeurs des *targets* binaires à prédire étant conservées par GENERALI.

La métrique retenue pour évaluer la précision avec laquelle les prédictions sont faites est le coefficient normalisé de Gini. La définition donnée pour le challenge est quelque peu différente de la version usuelle, et permet de comparer des probabilités à de vraies *targets*.

L'idée est de donner un meilleur score aux modèles qui détectent avec une probabilité plus importante les bâtiments $X_1, \dots, X_n$ qui ont effectivement fait une réclamation. Pour cela, les probabilités $P(X_1 = 1) = p_1, p_2, \dots, p_n$ sont triées pas ordre décroissant pour obtenir $p_{\sigma_p(1)} \geq p_{\sigma_p(2)} \geq \dots p_{\sigma_p(n)}$, et on calcule alors :

$$\mathrm{Gini}(a,p) = \frac{1}{n}\Big(\frac{\sum_{j=1}^{n}\sum_{i=1}^{j} a_{\sigma_p(i)}}{\sum_{j=1}^{n} a_{\sigma_p(j)}} - \frac{n+1}{2}\Big) \tag{1}$$

où a est le vecteur des *targets* réelles.

Le score final "normalisé" est alors défini par $\frac{\mathrm{Gini}(a,p)}{\mathrm{Gini}(a,a)}$.

Ainsi, un modèle qui attribue la même probabilité $p_1 = \ldots = p_n = 0.5$ à tous les bâtiments obtient un score de

$$\frac{1}{n}\Big(\frac{n+1}{2} - \frac{n+1}{2}\Big) = 0.$$

1.3 Données

Le jeu de données fourni contient les informations relatives à près de 11000 bâtiments assurés par GENERALI. On y retrouve un certain nombre de variables continues et catégorielles :

- `Identifiant` : identifiant du client Generali ;
- `ft_2_categ` : année de souscription à l'assurance ;
- `EXPO` : durée de couverture dans l'année (par exemple, `EXPO = 1` s'il s'agit d'une assurance sur toute l'année, 0.5 si c'est sur 6 mois) ;
- `superficief` : superficie de l'habitation ;
- `Insee` : code permettant de localiser géographiquement l'habitation ;
- `target` : variable catégorielle à prédire (0 s'il n'y a pas de déclaration de sinistre, 1 s'il y en a au moins une) ;
- `ft_i_categ (i = 4, ..., 24)` : variables catégorielles non définies, qui décrivent des caractéristiques de l'habitation.

On remarque que la plupart des données fournies sont des variables catégorielles non documentées. Cela ajoute ainsi une complexité supplémentaire au challenge, car nous ne pouvons pas avoir une approche intuitive sur ces données.

Le code Insee est éminemment important, dans la mesure où l'une des difficultés de ce challenge est de fusionner le jeu de données existant avec d'autres sources de données externes, afin d'améliorer autant que possible la précision de nos modèles.

La suite de notre rapport se structure de la sorte : la première partie traite de la phase de *data cleaning* et d'étude exploratoire des données. Nous nous attacherons dans

un second temps à détailler les différents algorithmes implémentés. Enfin, une analyse des résultats obtenus et des principales difficultés rencontrées sera faite.

2 Nettoyage et exploration des données

Trois jeux de données sont initialement fournis par GENERALI, à savoir :

— `x_train.csv` contenant près de 11000 lignes avec 26 colonnes portant sur des variables continues comme catégorielles. C'est sur ce jeu de données que nos modèles ont été entraînés ;
— `y_train.csv` contient les *targets* associées aux observations. L'objectif de ce challenge est d'inférer la probabilité avec laquelle chaque bâtiment est susceptible de déposer au moins une réclamation au cours de la période d'assurance ;
— `x_test.csv` : ce jeu de données présente exactement les mêmes variables que `x_train.csv` mais un nombre d'entrées réduit : environ 3400 observations. Une fois nos modèles entraînés sur le train set, c'est sur ce jeu de données que la précision de nos modèles sera évaluée. À noter cependant l'absence d'un jeu de données `y_test.csv`. Celui-ci est conservé par GENERALI pour comparer les prédictions de nos modèles sur `x_test.csv` avec les vraies valeurs de *target*, lors de chaque soumission de nos résultats sur la plateforme dédiée au challenge.

Nous nous concentrons dans un premier temps sur les données issues du train set, dont voici un aperçu :

	Identifiant	ft_2_categ	EXPO	ft_4_categ	ft_5_categ	ft_6_categ	ft_7_categ	ft_8_categ	ft_9_categ	ft_10_categ	...	ft_17_categ	ft_18_categ
0	18702	2014	1.000000	0	V	N	1	O	1	O	...	V	base
1	3877	2014	1.000000	0	V	V	V	V	V	V	...	N	base
2	4942	2013	1.000000	1	V	V	V	V	V	V	...	N	base
3	13428	2013	0.246575	0	N	V	V	V	V	V	...	N	base
4	17137	2015	1.000000	0	V	N	2	O	1	O	...	V	base

ft_19_categ	superficief	ft_21_categ	ft_22_categ	ft_23_categ	ft_24_categ	Insee	target
2	1351.0	4	2012.0	0.0	2	65440	0
2	1972.0	2	1980.0	0.0	.	14341	1
2	1630.0	4	NaN	0.0	.	75109	0
2	532.0	3	NaN	0.0	.	92004	0
2	1050.0	2	1972.0	0.0	4	59340	0

FIGURE 1 – Jeu de données d'entraînement

2.1 Données manquantes

Dans les 11 000 entrées de cette table, 1236 n'ont pas de valeur renseignée pour le champ `ft_22_categ`. Il en est de même pour `superficief` (119) et `Insee` (115). Notre stratégie diffère selon les champs :

— concernant la variable `superficief`, l'analyse descriptive nous révèle une moyenne et une médiane très proches. Au vu de l'écart-type assez élevé, on préfère imputer la médiane calculée sur l'ensemble des observations plutôt que la moyenne, même si cela comporte un risque ;
— pour la variable `ft_22_categ`, on procède de même ;
— enfin, pour la variable `Insee`, on procédera également par imputation sur les données que l'on ajoute à l'aide de jeu de données extérieurs.

Toutes ces tâches peuvent être réalisées à l'aide du `SimpleImputer` fourni par Scikit Learn.

2.2 One Hot Encoding

Nous avons recourt au One Hot Encoding afin de transformer nos variables catégorielles en vecteurs contenant un 1 pour la modalité renseignée, et des 0 partout ailleurs.

On procède comme suit pour traiter un dataframe complet.

```
cols = list(df.columns)
names = list(df.columns[df.dtypes == 'object'])

# pour chacun des types categoriels, on fait du one hot
transform_cat = [(name, OneHotEncoder(categories = "auto"), [cols.
    index(name)]) for name in names]

# le reste des variables n'est pas affecté
tr = ColumnTransformer(transform_cat, remainder = 'passthrough')
tr.fit(df.values)
```

Le `ColumnTransformer` a le mauvais goût de réordonner les colonnes, il est heureusement facile de retrouver les "slices" qui correspondent à l'encodage de nos variables catégorielles, en comptant le nombre de catégories.

```
sizes = [0]
sizes.extend([len(tr.named_transformers_[name].categories_[0]) for
    name in names])
sizes = np.array(sizes).cumsum()
slices = [np.s_[sizes[i]:sizes[i+1]] for i in range(len(sizes) - 1)]
```

Au final, en recollant avec les variables non encodées, on trouve

variable	emplacement	variable	emplacement
ft_5_categ	[0, 3[	ft_17_categ	[36, 39[
ft_6_categ	[3, 6[	ft_18_categ	[39, 44[
ft_7_categ	[6, 10[	ft_23_categ	[44, 51[
ft_8_categ	[10, 13[	ft_24_categ	[51, 62[
ft_9_categ	[13, 17[	ft_2_categ	62
ft_10_categ	[17, 20[	EXPO	63
ft_11_categ	[20, 23[	ft_4_categ	64
ft_12_categ	[23, 26[	ft_19_categ	65
ft_13_categ	[26, 29[	superficief	66
ft_14_categ	[29, 32[	ft_21_categ	67
ft_15_categ	[32, 34[	ft_22_categ	68
ft_16_categ	[34, 36[		

TABLE 1 – Emplacement des variables – certaines variables catégorielles sont traîtées comme numériques

2.3 Sélection de variables

Au vu des nombreuses variables catégorielles, le nombre de colonnes explose rapidement après encodage. On souhaite donc faire de la sélection de variables. Faire de la sélection après encodage est un peu délicat. Nous avons essayé quelques méthodes.

Avec `FunctionTransformer`. Il est possible de créer des prédicteurs par assemblage de plus petits éléments : des "transformers" qui modifient les données, suivis de prédicteurs plus classiques. Les prédicteurs ainsi construit peuvent être ajustés par validation croisée sur chacun des paramètres de ces composants. Dans notre cas, on utilise le `FunctionTransformer` : très générique, il peut appliquer une fonction quelconque aux données.

```
# pipeline de transformation qui fait la choses suivante :
# 1. appel à une étape "drop" qui va supprimer certaines features
# 2. regression logistique
pipeline = Pipeline([
    ("drop", FunctionTransformer(validate = False)),
    ("logreg", LogisticRegression(solver = 'lbfgs', max_iter = 1000))
])
```

On peut donc faire de la validation croisée sur cette fonction, parmi l'ensemble des fonctions supprimant une des variables catégorielles (toutes les colonnes qui y sont associées après encodage), définies à partir des "slices" déterminées plus haut.

```
def make_drop(i):
    # fonction de suppression de la slice i
    def drop(x):
        return np.delete(x, i, 1)
    return drop

dropfuns = [make_drop(slice_) for slice_ in slices]
```

Enfin :

```
params = dict(drop__func = dropfuns)
search = GridSearchCV(pipeline, params, return_train_score = False, cv
    = 3, n_jobs = -1)
search.fit(xtrain, ytrain)
```

On pourrait aller plus loin et tenter d'enlever plusieurs variables. On peut tout de même faire quelques remarques sur le résultat obtenu. Les scores affichés n'ont rien à voir avec les coefficients de Gini.

colonne supprimée	score moyen	variance	rang
ft_5_categ	0.782720	0.002931	13
ft_6_categ	0.784920	0.003575	3
ft_7_categ	0.782476	0.001887	15
ft_8_categ	0.783209	0.003419	10
ft_9_categ	0.784920	0.003575	3
ft_10_categ	0.783209	0.003419	10
ft_11_categ	0.784431	0.002961	7
ft_12_categ	0.785164	0.005081	1
ft_13_categ	0.785042	0.003138	2
ft_14_categ	0.782720	0.002182	13
ft_15_categ	0.784920	0.003575	3
ft_16_categ	0.784920	0.003575	3
ft_17_categ	0.782231	0.002036	16
ft_18_categ	0.783820	0.003166	9
ft_23_categ	0.784309	0.003959	8
ft_24_categ	0.783087	0.001210	12
(toutes)	0.781987	0.003020	17

Il semble qu'à première vue enlever toutes les variables catégorielles ne changerait que peu les choses (du moins en ce qui concerne la régression logistique).

Avant encodage, par test de Student. Avant encodage, on peut aussi faire des tests de Student en comparant les données pour deux modalités données d'une variable catégorielle. Pour une colonne donnée, avec `only_num` le dataframe `df` ne contenant que des données numériques :

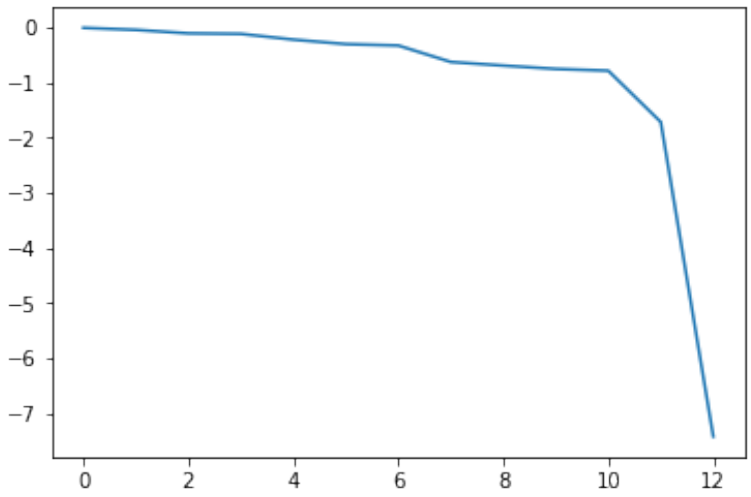
```
levels = df[column].astype('category').cat.categories
# pour chaque couple de niveaux
for level_a, level_b in combinations(levels, r = 2):
    # test de student : test.pvalue nous intéresse !
    test = ttest_ind(
        only_num[df[column] == level_a].values,
        only_num[df[column] == level_b].values,
        axis = None)
```

Toute la difficulté est ensuite dans la combinaison des p-values obtenues. On choisit de considérer le max de celles-ci, autrement dit : si la distribution des variables change significativement peu pour au moins deux modalités, on suppose que la variable catégorielle n'est pas intéressante avec le même niveau. C'est évidemment faux, mais c'est un critère satisfaisant pour tenter d'éliminer des variables.

On procède alors de manière "greedy" :

1. calcul des p-values ;
2. suppression de la variable catégorielle pour laquelle la p-value maximale (pour chaque couple de modalités) est la plus importante, sauf si la valeur obtenue est trop faible ;
3. retour en 1.

On voit alors qu'enlever toutes les variables catégorielles n'est pas recevable. Le tracé suivant indique (échelle log) la p-value associée à la variable supprimée à l'étape k : au delà de 10 ou 11 varuables supprimées, les valeurs des p-values s'effondrent.



Par ACP. Enfin, sur les données encodées, on peut procéder par ACP pour évaluer l'importance des variables.

On constate en premier lieu qu'une composante explique à elle seule près de la totalité de la variance.

```
>>> pca.explained_variance_ratio_
array([9.99810334e-01, 1.87595882e-04, ... ])
```

De plus, si l'on regarde de plus près le calcul de cette composante, on constate que les variables catégorielles n'ont qu'une importante marginale dans son calcul.

```
>>> np.arange(69)[pca.components_[0,:] > 0.001]
array([66, 68])
>>> pca.components_[0, pca.components_[0,:] > 0.001]
array([0.9999967 , 0.00256434])
```

D'après le tableau 1, sont importantes les variables `superficief` (66) et `ft_22_categ` (68).

2.4 Jeux de données externes

Nous nous sommes tout d'abord restreint aux variables fournies par GENERALI pour implémenter nos premiers modèles prédictifs. Nous avons cherché par la suite à enrichir nos jeux de données à l'aide de sources externes. Et c'est bien là tout l'enjeu de la variable Insee : celle-ci est censée servir de variable "jointure" pour fusionner plusieurs jeux de données, dans la mesure où un grand nombre de data sets ouverts au public disposent de ce champ.

Données économiques. Un premier jeu de données diffusé sur le site data.gouv.fr a retenu notre attention. Il contenait majoritairement des variables à connotation géographique (département), économique et social (PIB par habitant, revenu horaire moyen des ménages).

Cependant, il est trop difficile d'imputer toutes ces nouvelles variables pour les lignes ne disposant pas d'identifiant INSEE. En conséquence, nous nous sommes tournés vers des jeux de données plus simples pour lesquels il est possible de compléter les données.

Données météorologiques. Le second jeu de données qui nous a intéressé est fourni par opendatasoft.com. Celui-ci comporte notamment quelques variables catégorielles sur

des événements météorologiques exceptionnels (inondations, vents violents...), aux modalités "vert", "jaune" et "orange", groupées selon un département et pour une date donnée.

Le jeu de données est très volumineux – près de 4 Go de données (du fait de données géométriques décrivant la forme des départements). Il a donc fallu procéder à un premier traitement consistant à charger une ligne à la fois en mémoire et à supprimer les colonnes de données ne nous intéressant pas. Le fichier filtré pèse alors 1,7 Mo, ce que nous pouvons traiter sans problème avec Pandas.

Nous avons choisi de créer une seule variable à partir des données restantes : pour chaque code INSEE, on considère le nombre d'évènements n_V (resp. n_J, n_O) classés en "vert" (resp. "jaune", "orange"). On calcule alors un coefficient $\frac{0.5\times n_J+n_O}{n_V}$ qui est d'autant plus important que se produisent d'évènements exceptionnels pour un code INSEE donné. Cette valeur artificelle est plus aisée à imputer pour les données qui n'ont pas d'INSEE (on ajoute la valeur médiane).

Là encore, cependant, une rapide ACP nous montre que cette nouvelle variable n'a que peu d'importance... Comme auparavant, la première composante se détache nettement et le coefficient attribué à la nouvelle variable est négligeable.

3 Algorithmes et modèles

3.1 Le problème

Pour que le modèle puisse être évalué par le coefficient de Gini, il faut fournir un vecteur de probabilités, et non des étiquettes. Cela restreint quelque peu la famille de prédicteurs que l'on peut mettre en place.

3.2 Régression logistique binaire

On rappelle ici les fondements de la régression logistique binaire : disposant de n observations $(x_1, y_1), ..., (x_n, y_n)$ – avec $x_i \in \mathbb{R}^p$ et $y_i \in \{0, 1\}$ – on souhaite déterminer les probabilités

$$\pi = P(Y = 1) \quad \text{et} \quad 1 - \pi = P(Y = 0). \tag{2}$$

Pour cela, on introduit une fonction réelle monotone $f : [0, 1] \to \mathbb{R}$ et l'on cherche à

construire un modèle linéaire.

Plusieurs fonctions à l'allure sigmoïdale conviennent. En pratique, on retient la fonction *logit* définie par :

$$f(\pi) = \text{logit}(\pi) = \ln \frac{\pi}{1-\pi} \quad \text{avec} \quad f^{-1}(x) = \frac{e^x}{1+e^x}. \tag{3}$$

L'hypothèse de la régression logistique consiste ensuite à supposer que $\text{logit} \circ \pi$ est affine, ce qui permet ensuite de procéder par régression linéaire habituelle :

$$\hat{\pi}(X) = \frac{e^{\beta' X}}{1+e^{\beta' X}}. \tag{4}$$

3.3 XGBoost

XGBoost est une méthode d'apprentissage ensembliste dont les avantages ne sont plus à prouver : possibilité d'effectuer des calculs en parallèle sur une seule machine, optimisation des ressources en cache, traitement automatisé des valeurs manquantes, apprentissage continu pour *booster* un modèle déjà entraîné sur de nouvelles données, etc.

Formellement, il s'agit d'une méthode de boosting, où l'on construit un prédicteur comme une combinaison linéaire de prédicteurs plus simples. XGBoost se distingue dans sa façon d'optimiser itérativement les poids de la combinaison, en ajoutant un terme de régularisation.

4 Analyse des résultats

4.1 Sélection de variables

La plupart des méthodes de sélection que nous avons employé semble indiquer que deux variables sont particulièrement importantes à l'analyse, et que les variables catégorielles inconnues n'ont au final que peu d'intérêt. Il est dommage de ne pas en connaître le sens pour faciliter leur intégration aux modèles.

C'est donc aux variables de superficie et à `ft_22_categ` qu'il faut s'intéresser en premier lieu.

4.2 Modèles

Dans l'ensemble, XGBoost nous a fourni de meilleurs résultats. Il est cependant difficile d'évaluer précisemment la pertinence des paramètres choisis, puisque le "vrai" critère de performance – la soumission d'une solution au challenge – n'est accessible que de manière limitée. Un résultat jugé meilleur par validation croisée sur nos données s'est souvent avéré moins bon sur les données de test.

De plus, l'outillage de Scikit Learn facilite la validation croisée sur un résultat de classification ; il faut recourir à des méthodes un peu plus manuelles si l'on souhaite comparer les probabilités obtenues pour plusieurs modèles.

Les paramètres par défaut du XGBClassifier – 100 estimateurs, de profondeur maximale égale à 3 et avec un taux d'apprentissage fixé à 0.1 – sont donc ceux qui nous ont donné un meilleur score.