UCLouvain

Faculté des Sciences

LDATS2310 Data science for finance and insurance

Projet : Modélisation du nombre de sinistres en RC auto

Arthur Lacroix

Professeur :
Donatien Hainaut

Table des matières

1 Introduction

Dans ce travail, nous cherchons à prédire le nombre de sinistres automobiles sur base d'un portefeuille de polices d'assurances. Chaque colonne de la base de données correspondra à une variable explicative qui nous sera utile pour déterminer le nombre de sinistres estimé.

Les modèles présentés sont les suivants : GLM, CART, random forest, GBM et Neural Net. A des fins de reproductibilité des résultats, nous choisirons `set.seed(2)` dans les différents codes `R` présentés.

Une fois le code `R` de nos modèles écrit, nous les entraînerons grâce à notre base de données "DBtrain". Pour cela, nous créons une partition des données de cette base données. Nous prenons 90% des données afin d'entraîner notre modèle et nous réaliserons des prédictions sur les 10% restant. Nous effectuerons ensuite pour chaque modèle une cross validation de 10 sous ensembles afin de rendre notre modèle plus robuste.
Une fois les prédictions réalisées avec notre modèle, nous les comparerons avec les données observées. Nous pourrons alors calculer la déviance Poisson du modèle. Cette déviance étant calculée de la même manière pour touts nos modèles, nous pourrons alors les comparer entre eux et ainsi déterminer le plus performant. Nous choisirons ensuite ce modèle pour prédire le nombre de sinistre de la base de données "DBtest".

Tous les codes `R` sont disponibles sur le répertoire github `https://github.com/arlacroix/Data-science-LDATS2310.git` que nous avons jugé nécessaire de créer pour ce projet.
Nous précisions tout de même que les différents codes `R` présentés ici sont essentiellement inspirés des codes du professeur Donatien Hainaut dans le cadre du cours LDATS2310 : Data science for finance and insurance.

2 Modèle GLM

Nous allons dans cette section proposer de calculer le nombre espéré de sinistres automobiles à partir d'un portefeuille de polices d'assurances. Nous utiliserons ces données pour entraîner notre modèle et nous soumettrons ensuite une autre table de données afin de prédire le nombre de sinistres attendu.

Pour faciliter l'implémentation d'un modèle GLM dans le logiciel `R`, nous utiliserons le package `glmnet` et plus particulièrement la fonction `glm` de ce package.

Nous devons ensuite faire quelques modifications sur les fichiers de données que nous avons. Pour faciliter la lecture des données par notre modèle, nous créons plusieurs catégories au sein des variables "CarAge" et "DriverAge".

En ce qui concerne les catégories de la variable "CarAge", nous la diviserons en deux catégories : les véhicules âgés entre 0 et 5 ans et les véhicules âgés de plus de 5 ans. Nous avons choisi 5 ans car cela correspond à l'âge entier le plus proche de la médiane de cette variable (57,48% des personnes ont un véhicule âgé de moins de 5 ans et 42,52 % des personnes ont un véhicule âgé de plus de 5 ans).

Pour la variable "DriverAge", nous avons choisi plusieurs catégories d'âges : 15-28 ans, 29-31 ans, 32-35 ans, 36-44ans, 45-51 ans, 52-61 ans, 62-100 ans.

Remarque

Ces étapes préliminaires serons faites pour l'implémentation de chaque modèle.

Une fois ces étapes préliminaires terminées, nous utilisons la fonction `glm` afin d'implémenter facilement un modèle GLM.

Une autre manière d'implémenter le modèle GLM est d'utiliser la fonction `train` du package `caret`. Nous avons alors choisi d'implémenter les deux méthodes afin de voir laquelle est la plus performante. Nous verrons les résultats dans la dernière section de ce rapport.

3 Modèle CART

Nous utilisons dans cette section un autre modèle toujours pour déterminer le nombre de sinistres automobiles.

Nous commencerons par créer un arbre trop complexe (le coefficient de complexité, `cp`, utilisé n'est pas optimal) et nous chercherons ensuite un moyen de trouver le paramètre de complexité le plus adéquat possible afin de minimiser l'erreur de cross validation commise. Pour cela, nous utilisons la fonction `printcp` qui va nous permettre de trouver l'erreur de cross validation minimale et le `cp` correspondant à cette erreur minimale.

Lorsque nous appelons la fonction `printcp(d.tree)` voici le genre de résultats que nous obtenons :

```
Variables actually used in tree construction:
[1] Area      CarAge    Contract  DriverAge Fract     Gender    Leasing   Power

Root node error: 25006/70000 = 0.35722

n= 70000

            CP nsplit rel error  xerror      xstd
1    5.0094e-03      0   1.00000 1.00005 0.0097248
2    2.6703e-03      1   0.99499 0.99513 0.0096884
3    1.6084e-03      2   0.99232 0.99314 0.0096879
4    1.1403e-03      3   0.99071 0.99159 0.0096901
5    1.0953e-03      4   0.98957 0.99121 0.0096840
6    6.3513e-04      5   0.98848 0.99019 0.0096737
7    5.6670e-04      7   0.98721 0.99068 0.0097042
8    4.8573e-04      8   0.98664 0.99042 0.0096990
```

FIGURE 1 – Résultat de la fonction `printcp`

Nous devons alors maintenant choisir le `cp` de sorte à ce que le `xerror` soit le plus petit possible. Étant donné le caractère aléatoire de ce processus nous utilisons le code suivant afin de nous assurer de toujours prendre le `cp` optimal à chaque fois que nous lancerons le code :

```
cp.opt <-d.tree$cptable[which.min(d.tree$cptable[,"xerror"]),"CP"]
```

Même si dans notre cas, nous avons choisi de mettre `set.see(2)` afin de pouvoir reproduire nos résultats malgré l'aléa.
Ayant maintenant une méthode afin d'extraire un `cp` optimal, nous recalculons un arbre optimal en y incluant le `cp` précédemment trouvé. Nous trouvons ainsi une prédiction pour le nombre de sinistres de chaque personne présente dans le portefeuille.

En utilisant le package R `rpart.plot` nous obtenons arbre plus intuitif à comprendre :

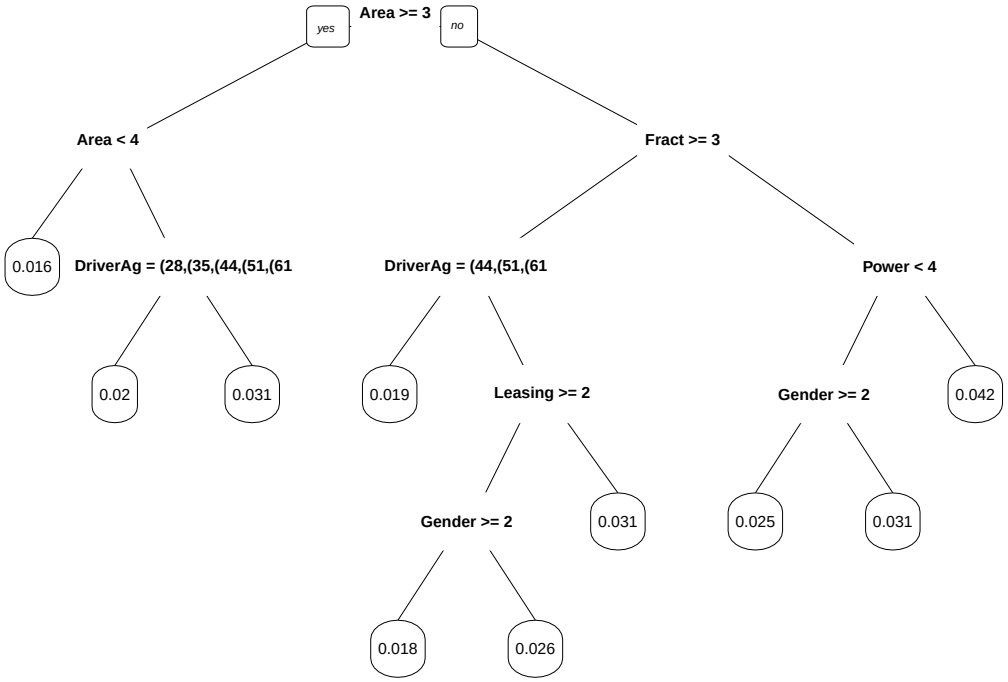

FIGURE 2 – Représentation graphique de l'arbre optimisé

Nous réutilisons alors la fonction `printcp` et nous avons cette fois-ci :

```
          CP nsplit rel error  xerror     xstd
1 0.00421650      0   1.00000 1.00014 0.010883
2 0.00279879      1   0.99578 0.99601 0.010843
3 0.00182391      2   0.99298 0.99331 0.010841
4 0.00156797      3   0.99116 0.99287 0.010844
5 0.00107710      4   0.98959 0.99096 0.010832
6 0.00095603      5   0.98852 0.99103 0.010844
7 0.00077561      6   0.98756 0.99040 0.010858
8 0.00057877      8   0.98601 0.99029 0.010867
9 0.00055097      9   0.98543 0.99177 0.010899
```

FIGURE 3 – Résultat de la fonction `printcp` sur l'arbre optimisé.

Nous voyons alors que, bien qu'ayant construit un arbre dont l'erreur de cross validation est minimum, la valeur du `xerror` ne décroît pas de manière significative an fonction du `cp`. Cela nous indique que notre arbre n'est malheureusement pas fiable.

Nous pouvons, comme pour le modèle GLM, utiliser la fonction `train` afin de voir quelle méthode d'implémentation est la meilleure.

On voit alors en comparant les éléments `cp` des deux arbres (celui optimisé et celui créer par la fonction `train`) que ceux-ci sont différents alors que nous avons commencer par prendre un `cp` identique. Cela signifie que la fonction `train` optimise le coefficient de complexité pour nous et nous verrons plus tard l'impact que cela a sur la déviance Poisson des deux différents arbres.

4 Modèle random forest

Ce modèle ne consiste plus à générer un seul arbre de régression mais bien plusieurs arbres afin de construire une "forêt" d'arbres. Intuitivement, on sent bien que le modèle random forest est censé être plus précis que le modèle CART qui ne considérait qu'un seul arbre de régression.

Nous commençons par créer une forêt qui n'est pas optimale et nous tenterons ensuite d'optimiser un peu plus notre modèle. On remarque qu'en prenant différent `cp` cela ne change pas fondamentalement les résultats mais que le temps de calcul augmente fortement. Nous ne jugeons dès lors pas nécessaire d'optimiser ce paramètre en sachant que les résultats sont quasiment identiques mais pour un temps de calcul beaucoup plus long.
Une autre méthode plus facile pour implémenter le modèle random forest est d'utiliser la fonction `randomForest` du package `randomForest`. Grâce à cette fonction, le code est beaucoup moins lourd et on s'aperçoit aussi que la fonction `randomForest` optimise pour nous le paramètre `cp`. Cela à toutefois un inconvénient majeur à savoir le temps de calcule. En effet, là où la première méthode ne mettait que quelques secondes (pour le code non optimisé) pour aboutir à un résultat, la deuxième met plusieurs minutes (environ 20 minutes). Nous verrons alors dans la section de comparaison des modèles dans quelle mesure cela à un impact sur la déviance Poisson.

Nous avons aussi voulu utiliser la fonction `train` avec la méthode `"rf"` afin d'avoir une troisième comparaison. Ce modèle ayant mis près de 5 heures à tourner avant de fournir un résultat, nous sommes en droit de penser que cette méthode est bien optimisée. Nous verrons alors dans la section où nous comparons les modèles si cette méthode est en effet plus efficace.

5 Modèle GBM

Dans cette section, nous tenterons d'implémenter un modèle GBM sur nos données. Pour cela, nous utiliserons la fonction `gbm` du package `gbm`.

Nous optimiserons ensuite le nombre d'arbre du modèle en utilisant la fonction `gbm.perf` qui nous donnera le nombre d'arbres optimal pour notre modèle.

On lance alors notre code `R` avec un modèle gbm contenant 50000 arbres. Après cela, nous utilisons la fonction `gbm.perf`. Celle-ci nous donne le résultat suivant

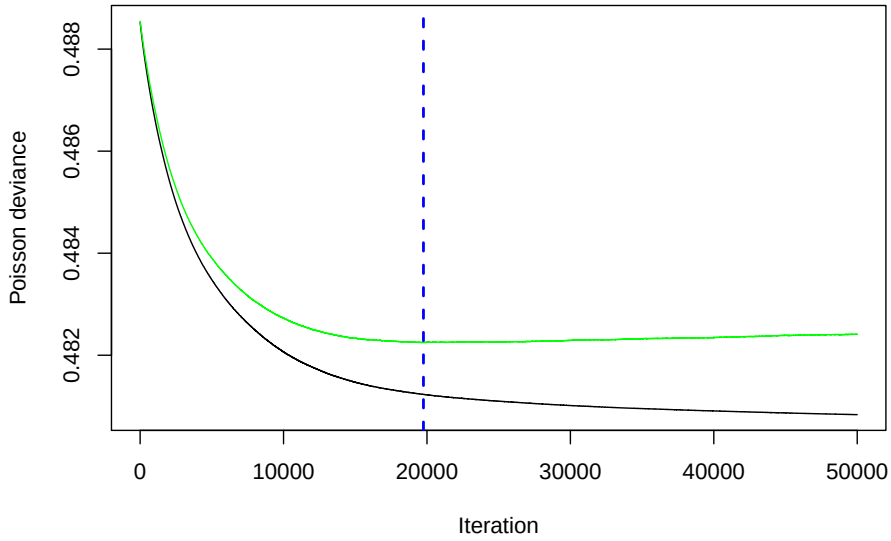

FIGURE 4 – Résultat de la fonction `gbm.perf` sur notre modèle GBM

Grâce à ce résultat, nous voyons que le nombre d'itération optimal est atteint pour 19754. Passer ce nombre d'itération, l'erreur du modèle augmente. Par conséquent, le modèle est optimisé avec 19754 arbres et donnera de meilleures prédictions. On note aussi que nous avons choisi le paramètre `shrinkage = 0.001` dans la fonction `gbm`. Le shrinkage peut être compris comme "le taux d'apprentissage" du modèle. Intuitivement, on sent bien que plus ce paramètre est petit, plus le modèle sera précis car faire beaucoup de petites étapes est mieux que d'en faire plusieurs grandes, mais nécessitera beaucoup plus d'arbres.

Nous essayons à présent de voir si notre modèle converge vers un nombre d'itération avec un shrinkage 100 fois plus petit, autrement dit, nous prenons `shrinkage = 0.00001` et nous lançons 50000 nouvelles itérations. Nous voyons que dans ce cas, le nombre optimal d'itération n'est pas encore atteint. Nous devons donc définir un nombre d'itération plus grand et relancer le modèle. Cela commence à devenir coûteux en terme de temps de calcule. En effet, à l'aide des commandes `tic` et `toc` on voit déjà que le temps de calcul pour 50000 itérations est proche d'une heure.
On pourrait alors améliorer le modèle en prenant un shrinkage vraiment proche de 0 mais avec un nombre d'arbres très élevé. Le gros problème sera alors le temps de calcul qui commencera à devenir vraiment long.

6 Modèle Réseau de neurones

Essayons de voir comment il nous est possible d'implémenter un réseau de neurones. Pour cela, nous reprenons le code `R` du professeur Donatien Hainaut disponible sur le site moodle du cours et nous le rendons compatible avec notre base de données.

Nous commençons par essayer le modèle sans ajuster les paramètres pour voir ce que cela donne. Nous constatons alors que le modèle est rapide (quelques secondes).

Les paramètres n'étant pas du tout optimisés, nous essayons simplement ici d'ajuster quelques

paramètres afin de voir comment le modèle réagit. Nous prenons alors les paramètres suivants : `treshold = 0.01`, `stepmax = 1e+06` et `lifesign.step = 1000`. Le paramètre `treshold` désigne le seuil de la dérivée partielle de la fonction erreur et sert de critère d'arrêt. Le deuxième, est simplement le nombre d'itération maximale que nous accordons à notre modèle afin de converger vers le `treshold` fixé. `stepmax` désigne simplement le nombre d'étapes faites avant d'imprimer dans la console l'état d'avancement du modèle. Ici par exemple, toutes les 1000 itérations nous imprimerons l'état d'avancement du modèle.

On voit alors qu'avec cette configuration, notre modèle converge après 248 414 itérations avec un temps de calcul de 2h06. Nous verrons dans la dernière section de ce rapport si cette configuration provoque une réelle différence sur la déviance Poisson du modèle.

7 Comparaison des modèles

Dans cette section, nous tenterons de comparer les modèles entre eux sur base de la déviance Poisson. Pour cela, nous utiliserons nos différents modèles créer précédemment sur nos données de validation que nous comparerons ensuite aux données observées. Nous pourrons alors calculer la déviance Poisson pour chaque modèle en utilisant la fonction `calc.deviance` du package `dismo` et comparer les modèles entre eux. Le modèle ayant la déviance Poisson la plus faible sera le plus performant étant donné que ses prédictions seront les plus proches de la réalité.

Nous avons dit précédemment que nous avons systématiquement implémenté les modèles de deux manières différentes (sauf pour le NeuralNet et le GBM). Les modèles implémentés de manières différentes que celle présentées au cours seront notées par `train(nom_du_modele)` lorsque la fonction `train` a été utilisée et par `randomForest(RF)` dans le cas du modèle random forest. Nous obtenons alors le tableau suivant

Modèle	Déviance Poisson	Déviance Poisson moyenne
`GLM`	2334.759	0.333537
`train(GLM)`	2467.554	0.3525077
`RT`	2509.039	0.3584341
`RT_opt`	2354.019	0.3362885
`train(RT)`	2495.076	0.3564394
`RF`	2457.032	0.3510046
`randomForest(RF)`	2484.537	0.3549339
`train(RF)`	2472.474	0.3532106
`GBM`	2795.45	0.39935
`NN`	2363.272154	0.3376103078
`NN_opt`	2374.797	0.3392568

La déviance Poisson a été calculée en prédisant le nombre de sinistres de notre validation set et en les comparant avec les valeurs observées du nombre de sinistres.

Nous voyons alors que le modèle ayant la déviance Poisson la plus faible est le modèle GLM implémenté avec la fonction `glm`. On voit également que la déviance du modèle CART est assez petite aussi mais nous avions vu que cette méthode n'était pas vraiment fiable du fait que l'erreur de cross validation restait importante même si le coefficient de complexité était faible. On choisit donc le modèle GLM pour faire nos prédictions.

Nous appliquons maintenant ce modèle sur la base de données "DBtest" afin d'obtenir le nombre de sinistres attendu par "catégorie" de personne. Ce nombre de sinistres est contenu dans un

grand vecteur λ. Ainsi, la première composante de ce vecteur correspondra au nombre de sinistres de la première personne de la base de donnée et ainsi de suite. Puisque le nombre de sinistres par individu est connu, il est aisé maintenant de connaître la fréquence des sinistres. En effet, il nous suffit simplement de diviser le nombre de sinistres par la variable `Exposure` qui correspond à la durée du contrat de l'assuré en question.

Nous tenons à préciser que lorsque nous utilisons la fonction `table(NbExposure)` nous voyons que certaines valeurs apparaissent plusieurs fois. Cela s'explique par le fait que ces valeurs sont propres à une catégorie de personne en particulier. Par exemple, toutes les personnes dans la catégorie : "Age :29-31, Gender :Male, Area :2, CarAge :5, Power :2, Fract :3, Leasing :1, Contract :1" auront la même valeur pour la fréquence de sinistre même si elles n'ont pas le même nombre de sinistres et la même durée du contrat. On voit donc que la durée du contrat n'influence pas la fréquence des accidents mais influence par contre le nombre d'accidents. En effet, sur un contrat de 5 ans la probabilité de sinistre est plus élevée que pour un contrat de 2 ans.

On crée alors une nouvelle colonne "Nbclaims" ainsi qu'une colonne "NbExposure" qui contiendront nos prédictions. Nous exporterons, grâce à la fonction `write.csv2`, notre base de données afin d'avoir un fichier .csv où les séparateurs seront des " ;" et dans lequel les nombre décimaux sont notés par ",".

En analysant maintenant nos prédictions, nous voyons que le nombre total de sinistres prédit est de 2065,91 pour 30000 assurés soit 6,8863 %. En guise de comparaison, nous avions initialement 4791 sinistres pour 70000 assurés soit 6,8442 %. On voit alors que nos résultats prédits sont proches de ceux observés.

Étant donné que le modèle le plus performant est le GLM, il peut être intéressant de voir quelles sont les variables qui ont le plus d'importance sur le nombre de sinistres. En utilisant alors la fonction `summaray(modnbclaims)` nous obtenons

```
Coefficients:
                  Estimate  Std. Error   z value               Pr(>|z|)
(Intercept)       -3.30913634  0.06934564 -47.71945 < 0.00000000000000222 ***
DriverAge(28,31]  -0.10422684  0.07063493  -1.47557            0.14005914
DriverAge(31,35]  -0.12524170  0.06004458  -2.08581            0.03699568 *
DriverAge(35,44]  -0.17252013  0.04612659  -3.74014            0.00018391 ***
DriverAge(44,51]  -0.25047666  0.05046510  -4.96336   0.000000692826657600 ***
DriverAge(51,61]  -0.30729764  0.05113897  -6.00907   0.000000001865901805 ***
DriverAge(61,100] -0.34831799  0.06040481  -5.76639   0.000000008098531547 ***
Gender2           -0.22342533  0.03187405  -7.00963   0.000000000002389467 ***
Area2              0.15730720  0.03730367   4.21694   0.000024764356279865 ***
Area3             -0.34428653  0.04500043  -7.65074   0.000000000000019983 ***
Area4             -0.02212802  0.04768426  -0.46405            0.64260981
Power2             0.15752147  0.04423269   3.56120            0.00036916 ***
Power3             0.18177610  0.04618706   3.93565   0.000082971732954472 ***
Power4             0.40384721  0.05627311   7.17656   0.000000000000714887 ***
CarAge(5,100]     -0.10683531  0.03097112  -3.44951            0.00056160 ***
Fract2            -0.15341223  0.04474590  -3.42852            0.00060688 ***
Fract3            -0.31428012  0.03687453  -8.52296 < 0.00000000000000222 ***
Leasing2          -0.22279661  0.03181216  -7.00351   0.000000000002496363 ***
Contract2          0.07026193  0.03586400   1.95912            0.05009854 .
Contract3          0.15738473  0.04326058   3.63806            0.00027470 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

FIGURE 5 – Variables importantes du modèle GLM

et les variables ayant le plus d'importance sont celles pour lesquelles il y a "***" sur le côte droit.

En effet, ces variables sont dites statistiquement significatives et ne sont donc pas négligeables.

On note aussi que nous aurions pu choisir de baser nos prédictions non plus sur le nombre de sinistres ou la fréquence de ceux-ci mais plutôt sur le coup de sinistres aussi appelé la sévérité. Il y a cependant quelques problèmes avec cette approche. En effet, généralement nous nous n'avons pas beaucoup de données sur les coûts de sinistres car parfois ceux-ci mettent plusieurs années avant d'être réglés. De plus, il est difficile d'expliquer l'importance des variables explicatives. Par exemple, on pourrait penser qu'une voiture âgée de 50 ans ne présentera pas un coût de sinistre élevé. Mais s'il s'agit en fait d'un ancêtre de collection, le sinistre pourra tout de même coûter très cher. Enfin, il est toujours difficile d'évaluer précisément le coût des sinistres de manière générale, cela varie trop d'une année à l'autre. Ce sont donc pour ces différentes raison que nous avons choisi d'évaluer le nombre de sinistres et leurs fréquences plutôt que leur sévérité.

8 Conclusion

En conclusion, nous avons su implémenter plusieurs modèles de machine learning afin de prédire le nombre de sinistres de notre portefeuille. Nous avons dans un second temps utiliser la déviance Poisson afin de comparer la différence entre nos prédictions et nos données observées sur notre base de données initiale. Nous avons utiliser cette méthode pour comparer les différents modèles entre eux pour en déduire que le plus performant est le modèle GLM.

Nous avons ensuite utilisé ce modèle sur notre base de données "DBtest" afin de prédire le nombre de sinistres de ce portefeuille et montré quelles étaient les variables explicatives les plus importantes d'après ce modèles.

Finalement, nous avons exporté nos résultats dans une nouvelle base de données en rajoutant simplement deux nouvelles colonnes à la base de données initiales.

On note que pour améliorer encore plus nos modèles, nous aurions pu faire une grille de paramètres à optimiser. Malheureusement l'implémentation des modèles est plus complexe et le temps de calcul augmente fortement. De plus, nous avions vu que, malgré l'implémentation différente des modèles (notamment avec des fonctions qui sont censés optimiser pour nous ces différents paramètres), le modèle GLM s'en sort le mieux.