0.1 Problème de zero excessifs

Dans de nombreux domaines, on souhaite modéliser des données de comptage, c'est-à-dire des variables qui représentent un nombre d'occurrences : nombre de sinistres, de visites, d'appels, de fraudes, etc. Ces données sont souvent modélisées à l'aide de distributions classiques comme la Poisson ou la binomiale négative.

Cependant, un problème fréquent survient : la proportion de zéros observés est bien plus élevée que ce que prédit un modèle de Poisson standard. Ce phénomène est connu sous le nom de zéro-inflation (zero-inflation), ou excès de zéros.

Prenons l'exemple d'une base de données de sinistralité en assurance automobile :

Une grande partie des assurés ne déclarent aucun sinistre au cours de l'année, tandis qu'un petit nombre peut en déclarer plusieurs. Ce déséquilibre produit un grand nombre de zéros qui ne peuvent pas être expliqués uniquement par la variabilité aléatoire d'un processus de Poisson.

Face à ce constat, les modèles classiques comme le modèle de Poisson peinent à reproduire fidèlement la forte proportion de zéros observée dans les données. En effet, ils cherchent à ajuster un unique taux moyen d'occurrence (lambda) pour l'ensemble des assurés. Résultat : pour accommoder les quelques individus ayant des sinistres, le modèle élève ce taux, ce qui réduit mécaniquement la probabilité attribuée aux zéros ($P(y = 0) = exp(-\lambda)$) et conduit à une sous-estimation des profils sans sinistre. Cela biaise les prédictions et surestime la sinistralité chez les bons profils.

Pour y remédier, on a recours à des modèles dits zéro-inflated, capables de distinguer les "vrais zéros structurels" (profils qui n'auront jamais de sinistre) des zéros aléatoires (absence de sinistre par hasard). C'est dans ce contexte que s'inscrit le Zero-Inflated Neural Network (ZINN) : une approche moderne qui combine la structure des modèles statistiques avec la flexibilité des réseaux de neurones.

0.2 Fonctionnement

Le Zero-Inflated Neural Network (ZINN) repose sur une idée simple mais puissante : au lieu de modéliser le nombre de sinistres par un unique processus, on décompose le problème en deux sous-tâches complémentaires, chacune prise en charge par une tête distincte du réseau.

La première tête, de nature logistique, cherche à prédire la probabilité que le sinistre soit structurellement nul. Elle produit une sortie comprise entre 0 et 1, notée p_zero, qui représente la probabilité que l'assuré appartienne à la "classe des bons profils", ceux qui n'auront jamais de sinistre. C'est une probabilité d'appartenance à la classe des zéros structurels.

La seconde tête modélise la fréquence des sinistres pour les profils qui ne sont pas dans cette classe structurelle. Elle s'apparente à une régression de Poisson classique, mais adaptée au cadre des réseaux de neurones. Elle produit un paramètre λ (lambda), strictement positif, représentant le taux attendu de

sinistres chez les assurés exposés. Ce lambda est souvent obtenu via une fonction d'activation de type Softplus, qui garantit la positivité tout en étant plus souple que l'exponentielle.

Ces deux têtes s'appuient sur des couches cachées partagées, qui transforment les variables d'entrée en un espace de représentation latent. L'idée est que le modèle extrait d'abord des facteurs de risque complexes et non linéaires, puis les exploite différemment selon la tâche : déterminer si l'individu est structurellement à zéro, ou estimer sa fréquence de sinistres conditionnellement à une exposition.

L'apprentissage repose sur une fonction de perte adaptée, qui combine les deux logiques :

Si l'observation réelle est zéro, le modèle maximise la probabilité d'un zéro soit structurel, soit issu du processus de Poisson ;

Si l'observation est strictement positive, le modèle apprend uniquement sur la partie Poisson, en supposant que l'individu est dans la population exposée.

Cette dissociation permet au ZINN de ne pas forcer les zéros dans le processus de Poisson, ce qui améliore la qualité des prédictions, en particulier lorsque les zéros sont massivement surreprésentés. C'est une architecture qui combine la rigueur des modèles statistiques zero-inflated et la souplesse des réseaux de neurones, capable d'apprendre des relations complexes dans des données hétérogènes.

0.3 Pourquoi ZINN au lieu de ZIP

Le modèle Zero-Inflated Poisson (ZIP) est une approche statistique bien connue pour traiter les données contenant un excès de zéros. Il combine une composante logistique (pour modéliser la probabilité d'avoir un zéro structurel) et une loi de Poisson (pour modéliser le nombre de sinistres chez les individus exposés). C'est un modèle interprétable, mais rigide : il repose sur des hypothèses linéaires, tant sur la relation entre les variables et la probabilité de zéro, que sur la relation entre les variables et le taux de sinistres λ.

Le Zero-Inflated Neural Network (ZINN) propose une alternative plus souple. Il conserve la même architecture conceptuelle que le ZIP (une composante logistique + une composante Poisson), mais remplace les relations linéaires par des réseaux de neurones profonds, capables de capter des interactions complexes, des effets non linéaires et des représentations latentes dans les données.

Cette différence devient cruciale dans les bases actuarielles modernes, souvent riches, avec des centaines de variables, des interactions implicites (par exemple entre âge, bonus-malus et type de contrat) ou encore des dimensions latentes issues de techniques comme les autoencodeurs. Le ZINN peut capter ces signaux faibles et les exploiter, là où un ZIP classique risque de les ignorer.

De plus, le ZINN permet une personnalisation de l'architecture : nombre de couches, activation, régularisation, etc., ce qui en fait un outil plus flexible pour s'adapter aux spécificités de la base de données. On peut aussi l'étendre facilement à d'autres cadres (Zero-Inflated Negative Binomial, modèles hiérarchiques, embeddings de variables catégorielles...).

En résumé :

Le ZIP est simple, rapide à estimer et interprétable, mais limité par ses hypothèses linéaires.

Le ZINN est plus complexe à entraîner, mais il s'adapte mieux à des structures de données riches, améliore la qualité des prédictions et permet d'analyser plus finement les profils à risque.

0.4 Mise en oeuvre

0.4.1 Définition du modèle Zero-Inflated Neural Network

Nous commençons ici par définir la classe ZeroInflatedNN, qui représente un modèle de réseau de neurones zéro-inflated. Ce modèle permet de prédire à la fois la probabilité de "zéros structurels" et le taux Poisson pour les observations non-nulles, afin de mieux gérer des données où les zéros sont surreprésentés et d'améliorer l'estimation des événements pour les observations positives.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class ZeroInflatedNN(nn.Module):
    def __init__(self, input_dim, hidden_dim=64):
        super(ZeroInflatedNN, self).__init__()

        # Shared hidden layers
        self.shared_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )

        # Zero-inflation head (logistic regression)
        self.zero_head = nn.Sequential(
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()  # Outputs p(zero)
        )

        # Count head (Poisson log-rate)
        self.count_head = nn.Sequential(
            nn.Linear(hidden_dim, 1),
            nn.Softplus()  # Ensures \lambda > 0
        )
```

```
29
30      def forward(self, x):
31          features = self.shared_net(x)
32          p_zero = self.zero_head(features).squeeze()  # Probability of zero
33          lambda_ = self.count_head(features).squeeze()  # Poisson rate \lambda
34
35          return p_zero, lambda_
```

— **Importation des bibliothèques :**

  — `torch` et `torch.nn` sont utilisés pour construire des réseaux de neurones en PyTorch.

  — `torch.optim` nous permet de définir un optimiseur pour l'entraînement du modèle.

  — `numpy` est également importé, bien qu'il ne soit pas directement utilisé dans ce code particulier, il pourrait l'être dans d'autres parties du projet.

— **Classe `ZeroInflatedNN` :**

  — Cette classe hérite de `nn.Module`, la classe de base pour tous les modèles dans PyTorch.

  — Elle a deux principales sorties : la probabilité d'un zéro structurel (`p_zero`) et le taux Poisson (`lambda_`).

— **Initialisation (`__init__`) :**

  — `input_dim` : C'est la dimension des entrées que le modèle recevra. Dans ce cas, il s'agit de l'ensemble des variables explicatives (par exemple, les variables latentes).

  — `hidden_dim` : Le nombre de neurones dans les couches cachées. Par défaut, il est de 64, mais il peut être ajusté en fonction de la complexité des données.

— **Couches cachées (`shared_net`) :**

  — Ce bloc contient des couches entièrement connectées (ou `linear layers`) qui transforment les entrées en représentations plus complexes.

  — La première couche transforme l'entrée de `input_dim` vers `hidden_dim`. Ensuite, `ReLU` est appliqué pour introduire de la non-linéarité.

  — Une deuxième couche `nn.Linear(hidden_dim, hidden_dim)` est ajoutée, suivie d'une autre activation `ReLU`.

— **Tête de Zéro-inflation (`zero_head`) :**

  — Cette partie du réseau prédit la probabilité que l'observation soit un zéro structurel.

  — `nn.Linear(hidden_dim, 1)` réduit la sortie des couches cachées à un seul neurone.

  — `nn.Sigmoid()` transforme cette sortie en une probabilité (comprise entre 0 et 1).

— **Tête de comptage (`count_head`) :**

— Cette partie prédit le taux Poisson λ pour les observations non-nulles.

— `nn.Linear(hidden_dim, 1)` réduit également la dimension à une seule valeur (la prédiction de λ).

— `nn.Softplus()` est utilisé pour garantir que λ soit toujours positif, car le taux de Poisson doit l'être.

— **Propagation avant (`forward`) :**

— Le modèle prend en entrée les données `x`, les fait passer par les couches cachées partagées (`shared_net`).

— Ensuite, il génère la probabilité de zéro (`p_zero`) et le taux Poisson (`lambda_`) à travers leurs têtes respectives.

— `squeeze()` est utilisé pour retirer les dimensions inutiles de la sortie, ce qui simplifie l'interprétation.

Choix des hyperparamètres :

— `hidden_dim` : La taille de la couche cachée (par défaut 64) est une valeur courante. Plus cette valeur est élevée, plus le modèle peut capturer des relations complexes entre les variables, mais cela peut entraîner un risque de sur-apprentissage si le modèle est trop complexe par rapport à la quantité de données disponibles.

— Activation ReLU et Softplus : Ces fonctions sont choisies pour leurs bonnes propriétés en apprentissage de réseaux de neurones, permettant une meilleure propagation des gradients et assurant que les sorties respectent les contraintes du modèle (positivité pour λ).

0.4.2 Définition de la fonction de perte

Dans cette partie, nous définissons la fonction de perte zinb_loss, qui combine la gestion de l'inflation des zéros et la modélisation de la distribution Poisson pour les autres valeurs. Cette fonction permet de calculer la perte du modèle ZINN en fonction des prédictions du taux Poisson (lambda_) et de la probabilité de zéro structurel (p_zero), en traitant séparément les cas où y_true est nul ou non nul.

```
def zinb_loss(y_true, p_zero, lambda_, eps=1e-8):
    # Cas 1 : y = 0 (zéro-inflation + Poisson(0))
    loss_zero = -torch.log(p_zero + (1 - p_zero) * torch.exp(-lambda_) + eps)

    # Cas 2 : y > 0 (probabilité Poisson uniquement)
    loss_count = -torch.log((1 - p_zero) + eps) + nn.PoissonNLLLoss(log_input=False)(
    lambda_, y_true)

    # Combinaison des pertes (seulement appliquer la perte de comptage lorsque y > 0)
```

```
    loss = torch.where(y_true == 0, loss_zero, loss_count)
    return loss.mean()
```

— **Cas 1 : $y = 0$ (inflation de zéros + Poisson(0)) :**

— Si la valeur réelle y est égale à 0, la fonction de perte tient compte de l'inflation des zéros et de la probabilité de zéro structurel prédite par le modèle.

— La formule utilisée est :

$$\text{loss\_zero} = -\log(p_{\text{zero}} + (1 - p_{\text{zero}}) \cdot e^{-\lambda} + \epsilon)$$

où p_{zero} est la probabilité de zéro prédite et λ est le taux Poisson estimé. Le terme ϵ est ajouté pour éviter une division par zéro.

— **Cas 2 : $y > 0$ (vrais sinistres, Poisson) :**

— Si y est supérieur à 0, la fonction de perte calcule la log-vraisemblance d'une distribution Poisson avec le taux λ estimé.

— La formule utilisée est :

$$\text{loss\_count} = -\log(1 - p_{\text{zero}} + \epsilon) + \text{PoissonNLLLoss}(\lambda, y_{\text{true}})$$

où la perte de log-vraisemblance Poisson est calculée avec la fonction `PoissonNLLLoss`.

— **Combinaison des pertes :**

— La perte totale est calculée en combinant les deux pertes précédentes, selon que la valeur réelle y_{true} est égale à zéro ou non :

$$\text{loss} = \text{torch.where}(y_{\text{true}} == 0, \text{loss\_zero}, \text{loss\_count})$$

— La fonction renvoie la moyenne de la perte totale sur l'ensemble des exemples :

$$\text{loss.mean()}$$

0.4.3 Fonction d'entraînement : train_zinn

La fonction train_zinn permet d'entraîner un modèle Zero-Inflated Neural Network (ZINN). Elle prend en entrée le modèle, les données d'entraînement, le nombre d'époques d'entraînement, et le taux d'apprentissage. À chaque époque, elle effectue une passe avant pour prédire les sorties du modèle, calcule

la perte, effectue une rétropropagation et met à jour les paramètres du modèle. Le processus est répété pendant un certain nombre d'époques spécifié.

```python
def train_zinn(model, X_train, y_train, epochs=100, lr=0.01):
    optimizer = optim.Adam(model.parameters(), lr=lr)

    for epoch in range(epochs+1):
        optimizer.zero_grad()  # Réinitialisation des gradients
        p_zero, lambda_ = model(X_train)  # Passe avant du modèle
        loss = zinb_loss(y_train, p_zero, lambda_)  # Calcul de la perte
        loss.backward()  # Rétropropagation des gradients
        optimizer.step()  # Mise à jour des paramètres

        if epoch  10 == 0 :print(f"Epoch epoch, Loss : loss.item(:.4 f}")

    return model
```

- **Initialisation de l'optimiseur :** L'optimiseur utilisé pour la mise à jour des paramètres du modèle est `Adam`, avec un taux d'apprentissage spécifié par `lr`.
- **Boucle d'entraînement :** La boucle principale parcourt un nombre d'`epochs` (époques) données, et à chaque époque, plusieurs opérations sont effectuées :
  - **Rétropropagation et mise à jour des paramètres :** Avant chaque calcul, on appelle `optimizer.zero_grad()` pour réinitialiser les gradients de tous les paramètres du modèle. Ensuite, une passe avant est effectuée en appelant `model(X_train)`, ce qui génère les sorties prédites : la probabilité de zéro `p_zero` et le taux de Poisson `lambda_`.
  - **Calcul de la perte :** La perte est calculée avec la fonction `zinb_loss`, qui prend les vraies valeurs (`y_train`), la probabilité de zéro (`p_zero`), et le taux Poisson (`lambda_`) comme arguments. Cela permet de calculer l'erreur entre les prédictions du modèle et les véritables observations.
  - **Rétropropagation :** `loss.backward()` calcule les gradients de la fonction de perte par rapport aux paramètres du modèle.
  - **Mise à jour des paramètres :** `optimizer.step()` met à jour les paramètres du modèle en fonction des gradients calculés.
- **Affichage de la perte :** Toutes les 10 époques, la fonction affiche la valeur actuelle de la perte afin de suivre l'évolution de l'entraînement. Cela peut aider à déterminer si le modèle converge bien et si la perte diminue au fil du temps.
- **Retour du modèle :** À la fin des époques, la fonction retourne le modèle entraîné.