



**INSTITUT  
DE LA  
COMMUNICATION**



# Machine Learning

## TD - Imbalanced Learning

M2 Informatique - BI&A (2022-2023)

Guillaume Metzler

Institut de Communication (ICOM)

Université de Lyon, Université Lumière Lyon 2

Laboratoire ERIC UR 3083, Lyon, France

[guillaume.metzler@univ-lyon2.fr](mailto:guillaume.metzler@univ-lyon2.fr)

### Abstract

Dans ce TD, nous allons concentrer sur les techniques d'apprentissage qui sont spécifiques aux données déséquilibrées. Nous allons voir quels sont les différentes méthodes, sur le plan pratique, que nous pouvons employer afin de traiter cette problématique, permettant ainsi la construction d'algorithmes efficaces.  
Plus précisément, nous étudierons des méthodes liées à l'échantillonnage des données ou encore la pondération des erreurs effectuées par un algorithme (quand cela a un sens).

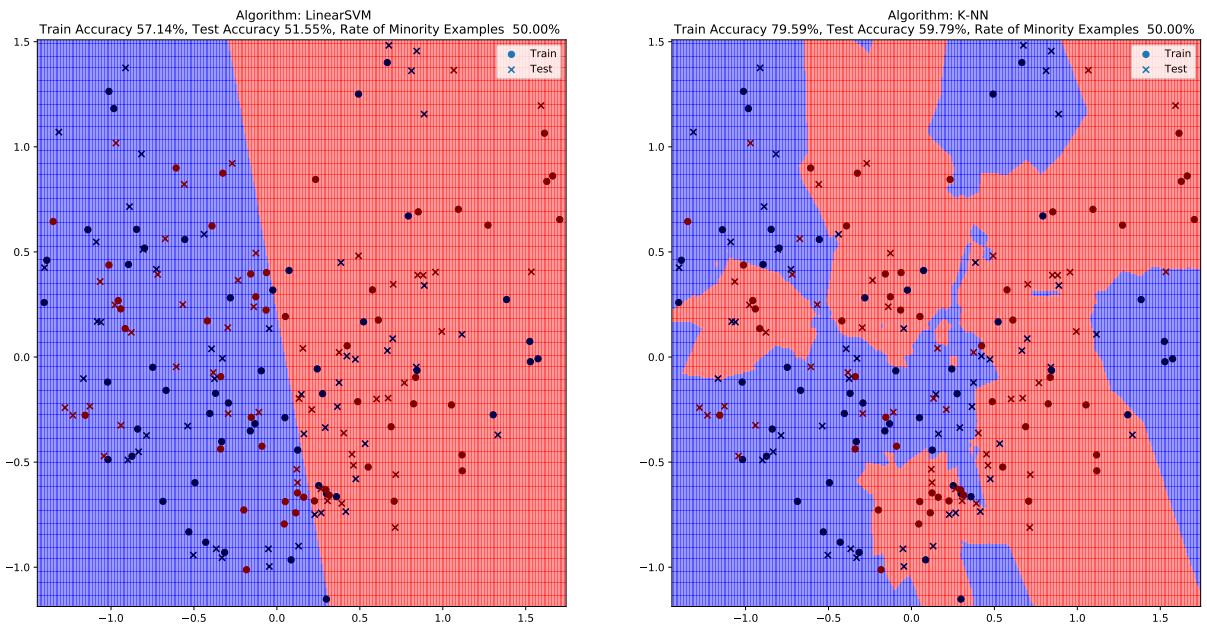


Figure 1: Illustration des performances d'un SVM linéaire (à gauche) et d'un algorithme du plus proche voisin (à droite) dans le cas où les classes en présence sont parfaitement équilibrées.

## 1 Imbalanced Learning

### 1.1 Contexte

Habituellement, lorsque nous travaillons avec des algorithmes d'apprentissage, notamment dans le cadre de la classification, les classes en présence sont représentées de façon équilibrées, *i.e.* nous avons en général le même nombre d'exemples dans chaque classe. Cet équilibre fait que la minimisation d'un taux d'erreur ou encore d'un substitut du taux d'erreur se révèle pertinent. L'exemple de la Figure 1 montre que les frontières des algorithmes testés : un SVM et un plus proche voisin, sont bien définis permettant de définir des zones de prédictions *positives* (rouge) et des zones de prédictions *négatives* (bleu).

Malheureusement dans certains contextes, cet équilibre n'est pas présent dans les données, une classe sera très souvent *sous représentée* (elle sera dite **minoritaire**) par rapport à l'autre classe (qui sera dite **majoritaire**). On parle alors d'*apprentissage dans un contexte déséquilibré*.

C'est un contexte que l'on rencontre très souvent dans les problématiques de *détection de fraudes*, comme la fraude fiscale (DGFiP), la fraude bancaire, les problèmes d'intrusions dans des réseaux sécurisés, dans un contexte de diagnostique médical pour la détection de maladies rares, etc. dont les enjeux peuvent avoir une très grande im-

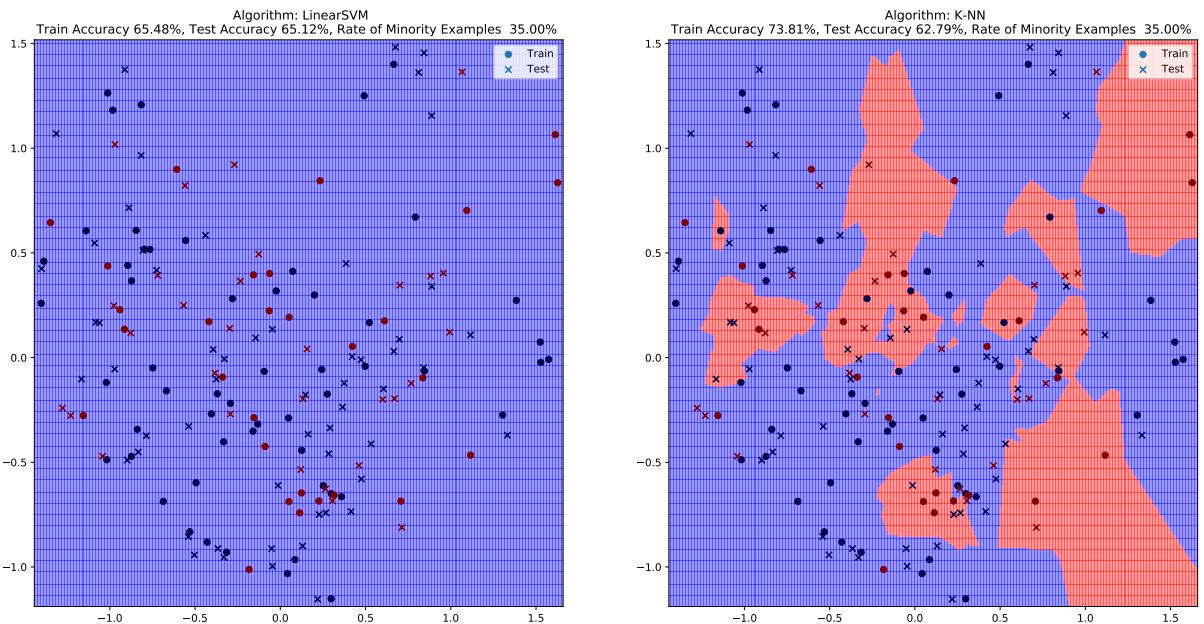


Figure 2: Illustration des performances d'un SVM linéaire (à gauche) et d'un algorithme du plus proche voisin (à droite) dans le cas où les classes en présence sont déséquilibrées.

portance sur les plans politique, économique, sociaux ou encore dans le domaine de la santé [Abdallah et al., 2016, Aggarwal, 2017].

Dans ces contextes là, la plupart des algorithmes standards sont inefficaces, comme le montre la Figure 2 reprenant nos deux précédents algorithmes. En effet, dans le cas du SVM, nos deux algorithmes ont tendance à prédire tout nos algorithmes comme appartenant à la classe négative. On peut également voir que la frontière de décision du plus proche voisin se réduit drastiquement.

1. Expliquer pourquoi on observe cet écart entre les deux situations ? Quelles sont les causes possibles au phénomène observé ?
2. Est-ce que l'on sera amené à observer le même type de comportement pour n'importe quel jeu de données ?

## 1.2 Mesures de performances

La mesure de performance classique consistant à évaluer le taux d'erreur de notre algorithme n'est plus pertinente dans ce type de contexte. Il faudra trouver des mesures qui sont capables de prendre en compte le déséquilibre dans notre jeu de données. Pour cela, on considère la matrice de confusion de notre algorithme, représenté en Table

	$h(\mathbf{x}) = 1$	$h(\mathbf{x}) = -1$
$y = 1$	Vrai positif	Faux négatif
$y = -1$	Faux positif	Vrai négatif

Table 1: Matrice de confusion d'un classifieur binaire

Cela pourra se faire en prenant en compte les erreurs effectuées sur la classe majoritaire mais aussi sur la classe minoritaire, ce que l'on peut faire avec la *Balanced Accuracy*:

$$BA = \frac{1}{2} \frac{TP}{TP + FN} + \frac{1}{2} \frac{TN}{TN + FP}.$$

Vous avez déjà rencontré le premier terme  $\frac{TP}{TP + FN}$  sous le nom de *sensibilité* ou encore de *rappel* de votre algorithme, *i.e.* on mesure la capacité du modèle à retrouver des exemples de la classe minoritaire. Le deuxième terme est connu sous le nom de *spécificité* qui mesure le taux de bonne prédictions négatives. La Balanced Accuracy se présente alors comme la moyenne de ses deux quantités.

On pourra également considérer une autre mesure de performances très courante dans le domaine, que l'on appelle la **F-mesure** et qui dépend d'un paramètre  $\beta$

$$F_\beta = \frac{(1 + \beta^2) \text{Rappel} \times \text{Précision}}{\beta^2 \times \text{Précision} + \text{Rappel}},$$

où le rappel a été défini précédemment et la précision, définie par

$$\text{Précision} = \frac{TP}{TP + FP},$$

représente la confiance du modèle dans ses prédictions positives, *i.e.* le taux de bonnes prédictions positives.

1. Evaluer la valeur de la Balanced Accuracy dans le cas où toutes les données sont prédites négatives lorsque la classe minoritaire représente 35% du jeux de données.
2. Exprimer la F-mesure en fonction des quantités TP, FN et FP.
3. Expliquer le rôle du paramètre  $\beta$  dans la définition de la F-mesure.
4. On considère les matrices de confusion obtenues à l'aide de deux hypothèses (modèles)  $h_1$  et  $h_2$  respectivement.

TP = 3	FN = 7
FP = 0	TN = 990

TP = 9	FN = 1
FP = 3	TN = 987

- (a) Evaluer l'accuracy, la Balanced accuracy et la F-mesure pour ces deux hypothèses.
- (b) Qu'elle est selon vous la meilleure hypothèse que l'on puisse considérer dans un contexte déséquilibré. Justifier votre réponse.

Il existe également d'autres mesures qui sont analogues à celles présentées précédemment, vous pourrez regarder chez vous de quoi il s'agit.

A noter que pour certains problèmes, notamment les problèmes de *ranking* (ou d'ordonnancement) il est souvent d'usage d'employer des mesures comme l'*AUC ROC* ou *Précision-Rappel* ou de mesurer des *Précision@k* ou encore *Rappel@k* sous des contraintes de traitement et toutes les autres qui en dérivent.

Si le choix d'une mesure de performance adaptée est importante dans ce contexte, elle ne permet de traiter le problème sur le plan algorithmique. Nous allons étudier ici deux méthodes dites de *pre-process* sur les données qui permettent de traiter le déséquilibre :

- les méthodes d'échantillonnage
- la pondération des erreurs

## 2 Échantillonnage

Nous avons vu que problème principal des problèmes déséquilibré réside dans le nombre de représentant de chaque classe et plus précisément dans le fait qu'une classe soit sous représentée par rapport à une autre. Pour traiter cet inconvénient au niveau des données, on peut chercher à rééquilibrer les classes en présence en

- en augmentant le nombre d'exemples de la classe minoritaire, on parle de *sur-échantillonnage*.
- en diminuant le nombre d'exemples de la classe majoritaire, on parle de *sous-échantillonnage*.
- en combinant les deux approches

## 2.1 Sous-échantillonnage

Dans cette première stratégie d'échantillonnage, on va chercher à rééquilibrer le jeu de données en réduisant le nombre d'exemples dans la classe majoritaire. Nous pouvons faire cela de différentes façons.

**Aléatoirement.** Le processus est simple et consiste à retirer aléatoirement un certain pourcentage d'exemples de la classe majoritaire de notre jeu de données.

On pourra faire cela avec la fonction *RandomUnderSampler* de la librairie **imblearn.under\_sampling** de Python. On pourra directement contrôler le déséquilibre final après échantillonnage.

**Condensed Nearest Neighbor.** Cette méthode initialement présentée comme moyen de réduire le nombre d'exemples à considérer pour l'algorithme des plus proches voisins peut aussi s'interpréter comme une méthode d'échantillonnage, plus précisément en supprimant un certain nombre d'exemples de la classe majoritaire et en conservant ceux de la classe minoritaire.

On pourra faire cela avec la fonction *CondensedNearestNeighbor* de la librairie **imblearn.under\_sampling** de Python.

**Tomek Link.** Il s'agit d'une version améliorée de l'approche précédente proposée par Ivan Tomek [Tomek, 1976]. Deux exemples de classe opposée  $\mathbf{p}_i$  et  $\mathbf{n}_i$  forment un lien *Tomek* si les conditions suivantes sont vérifiées :

- $\mathbf{p}_i$  est le plus proche voisin de  $\mathbf{n}_i$
- $\mathbf{n}_i$  est le plus proche voisin de  $\mathbf{p}_i$

On va ensuite supprimer de tels exemples de notre jeu de données, car ces derniers se trouvent à la frontière décision et peuvent vraisemblablement perturber l'apprentissage du classifieur. L'idée étant d'obtenir une frontière de décision plus nette, avec le moins de bruit possible à sa frontière. Dans le cas présent, on se contentera de supprimer uniquement ceux de la classe **majoritaire**.

On pourra faire cela avec la fonction *TomekLinks* de la librairie **imblearn.under\_sampling** de Python.

**Edited Nearest Neighbor.** Il s'agit d'une version très proche de Tomek Link [Wilson, 1972] L'idée est de se fixer un paramètre  $k$  qui correspond au nombre de voisins que l'on va considérer pour le process de nettoyage (en général  $k = 3$ ). Pour chaque exemple de la classe majoritaire on va :

- déterminer ses  $k$ - plus proches voisins
- déterminer la classe majoritaire dans le  $k$  voisinage
- supprimer l'exemple de l'ensemble d'apprentissage si la prédiction ne coïncide pas avec le vrai label

On pourra faire cela avec la fonction *EditedNearestNeighbours* de la librairie `imblearn.under_sampling` de Python.

**One Sided selection.** Cette dernière approche consiste à appliquer l'algorithme de *Condensed Nearest Neighbor* suivi ensuite par *Tomek Link* afin de réduire le temps de calcul de l'approche Tomek link lorsque les jeux de données sont volumineux.

On pourra se contenter d'appliquer les deux approches successivement ou encore d'utiliser la fonction *OneSidedSelection* de la librairie `imblearn.under_sampling` de Python.

Il existe bien sûr d'autres algorithmes de nettoyage et vous pourriez également créer votre propre procédure !

1. Quel est l'inconvénient de faire cela aléatoirement ?
2. Quel est l'intérêt de sous-échantillonner sur le plan algorithmique ? Quel est son inconvénient principal ?
3. Quel reproche pourrait-on faire concernant les méthodes "raffinées" de sous-échantillonage qui sont présentées ?

## 2.2 Sur-échantillonnage

Cette fois-ci on va chercher à **duplicer** voire à **synthétiser** des exemples de la classe minoritaire afin d'augmenter le nombre de ces exemples dans le jeu de données. A nouveau nous pouvons faire cela de plusieurs façons différentes et une liste non-exhaustive de ces méthodes est disponible [Fernández et al., 2018] mais nous nous contenterons de présenter les plus usuelles.

**Input:** Echantillon  $S$  avec  $N$  et  $P$ ,  $k$ : nombre de voisins,  $R$ : taux échantillonnage

$N = R \times P$ : Nombre d'exemples à créer

Choisir  $N$  exemples  $\mathbf{x}_i$  parmi  $P$

**for all**  $i$  in  $1:N$  **do**

    trouver les  $k$ -NN de  $\mathbf{x}_i$

    en choisir un aléatoirement  $\mathbf{z}_i$

**for all**  $j$  in  $1:\text{dim}(\mathbf{z}_i)$  **do**

        choisir  $\alpha \in [0, 1]$

        set  $[j] = \alpha \mathbf{x}_i[j] + (1 - \alpha) \mathbf{z}_i[j]$

**end for**

**end for**

**Output:** Un échantillon  $S' = 0$

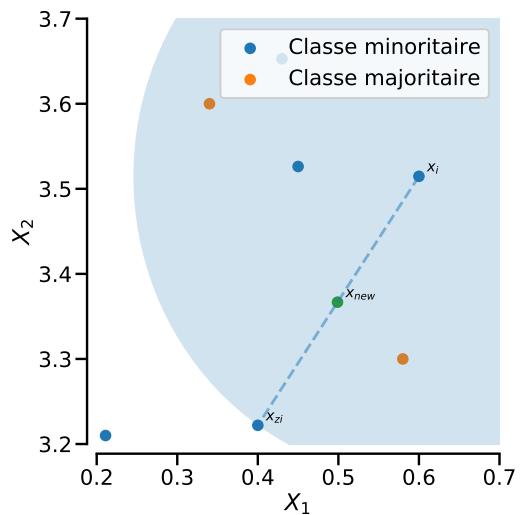


Figure 3: Algorithme *SMOTE* (à gauche) et illustration du procédé (à droite).

**Aléatoirement.** Le processus est simple et consiste à retirer aléatoirement un certain pourcentage d'exemples de la classe majoritaire de notre jeux de données.

On pourra faire cela avec la fonction *RandomOverSampler* de la librairie **imblearn.under\_sampling** de Python. On pourra directement contrôler le déséquilibre final après échantillonnage.

**SMOTE.** Il s'agit du premier algorithme proposé pour synthétiser des exemples appartenant à la classe minoritaire. Sur le plan pratique, on va construire un nouveau point sur le segment (ou l'hyper-cube selon la description qui en est faite, historiquement [Chawla et al., 2002] on fait cela sur un hyper-cube et c'est la description adoptée ci-dessous) qui relie deux exemples de la classe minoritaire qui sont "proches".

Une présentation de la procédure et une illustration sont disponibles en Figure 3.

On pourra faire cela avec la fonction *SMOTE* de la librairie **imblearn.under\_sampling** de Python. On prendra soin de préciser le nombre de voisins que l'on souhaite employer (par défaut  $k = 5$ ) mais aussi l'équilibre final désiré.

**Attention ! L'implémentation proposée est différente de la procédure présentée en Figure 3.**

**BorderSMOTE.** Elle repose sur l'algorithme SMOTE mais ce dernier ne sera appliqué que sur les exemples qui sont considérées comme difficiles à classer, *i.e.* ceux qui

se trouvent à la frontière de décision. Pour déterminer de tels exemples, on appliquer un  $k$ -NN, en général avec  $k = 3$ , et retenir ceux pour lesquels la prédiction ne coïncide pas avec le vrai label. Sur ces exemples là est ensuite appliqué un algorithme *SMOTE* pour construire un nombre fixé par l'utilisateur d'exemples de la classe minoritaire.

On pourra faire cela avec la fonction *BorderlineSMOTE* de la librairie **imblearn.under\_sampling** de Python.

**Safe Level SMOTE.** Très semblable à la méthode précédente, en revanche, on ne laisse pas la main à l'utilisateur sur le nombre d'exemples à générer selon la position de l'exemple positif. Ce nombre est automatiquement déterminé selon le niveau de sûreté de l'exemple. Ce niveau est défini comme le ratio d'exemples de la classe minoritaire dans le  $k$ -voisinage, plus cet indice est faible, plus on génère d'exemples.

**Adasyn.** Dans la même lignée que les approches précédentes, on va cette fois-ci mettre une distribution sur les exemples de la classe minoritaire. Cette distribution va nous permettre de fixer le nombre d'exemples générés via SMOTE sur l'ensemble des exemples de la classe minoritaire.

On pourra faire cela avec la fonction *ADASYN* de la librairie **imblearn.under\_sampling** de Python.

1. Quel est l'inconvénient de dupliquer ou de synthétiser des exemples sur le plan algorithmique ?
2. La synthèse via *SMOTE* est-elle toujours pertinente ? Illustriez rapidement vos propos
3. Si on dispose de 100 exemples dans la classe minoritaire et de 900 exemples dans la classe majoritaire.
  - (a) Comment fixer le paramètre *float* de la fonction python afin d'obtenir un équilibre entre les deux classes ?
  - (b) Comment fixer le paramètre *float* de la fonction python afin d'obtenir un équilibre de 20% - 80% ?
4. Quel est l'avantage de BorderSMOTE par rapport à SMOTE ?

### 2.3 Mise en pratique

Sélectionnez des jeux de données déséquilibrés parmi ceux que vous avez pu utiliser lors de la première séances pratique et tenter de mettre en oeuvre les approches précédentes.

	$h(\mathbf{x}) = 1$	$h(\mathbf{x}) = -1$
$y = 1$	$c_{TP}$	$c_{FN}$
$y = -1$	$c_{FP}$	$c_{TN}$

Table 2: Matrice de coût

On pourra par exemple comparer les résultats obtenus entre un algorithme dans sa version standard, lorsqu'il est combiné à une méthode sous-échantillonnage, lorsqu'il est combiné à du sur-échantillonnage, ou aux deux.

### 3 Pondération des erreurs

Cette deuxième approche consiste à modifier la façon dont les erreurs sont prises en compte par notre algorithme [Elkan, 2001]. On va partir du principe que, contrairement au principe habituel, les erreurs n'ont pas le même poids.

Si on souhaite faire en sorte que notre algorithme se focalise sur les exemples de la classe minoritaire, il va falloir lui indiquer que mal classer un tel exemple va entraîner une valeur importante de la loss, et que, au contraire, s'il classe mal un exemple de la classe majoritaire, cela aura une conséquence moindre.

La pondération ainsi effectuée va se faire en définissant au préalable une matrice de coût comme cela présentée en Table 2. Elle est semblable à une matrice de confusion mais, au lieu de regarder le nombre de bonnes/mauvaises décisions prises par l'algorithme, on va associer des poids/coûts aux décisions prises par ce dernier.

Dans notre contexte actuel, on va surtout se concentrer sur les valeurs de  $c_{FN}$  et  $c_{FP}$  qui définissent le poids des erreurs effectuées sur la classe minoritaire et majoritaire respectivement.

Par exemple, si on fixe  $c_{FN} = 2$  et  $c_{FP} = 1$ , alors mal classer un exemple de la classe minoritaire aura un coût deux fois plus élevé que de mal classé un exemple de la majoritaire. En procédant ainsi, on sera en mesure d'accorder plus d'importance à la classe minoritaire.

Dans les implémentations standards, nous avons toujours  $c_{FN} = c_{FP} = 1$ , *i.e.* toutes les erreurs ont le même poids.

Il n'y a pas d'implémentation spécifique pour le cost-sensitive learning, il faudra le plus souvent régler le paramètre 'class\_weight' des différents implémentations sous Python.

1. Supposons que mon jeu de données comporte 10% d'exemples dans une classe et 90% dans l'autre. Comment puis-je régler les poids des différentes classes afin

d'obtenir un coût global équilibré entre les deux classes.

2. A votre avis, est-ce toujours judicieux d'adopter le paramétrage précédent ? De quoi cela peut-il dépendre ?
3. Pourrions nous aller un cran plus loin dans la définition des coûts, *i.e.* la définir à une échelle plus fine ?

## 4 Pour finir

On tentera de mettre en pratique les différentes approches présentées dans le cadre de ce document et de déterminer laquelle semble la plus appropriée, entre échantillonnage et pondération des erreurs, en prenant le soin de bien choisir sa pondération.

## References

- [Abdallah et al., 2016] Abdallah, A., Maarof, M. A., and Zainal, A. (2016). Fraud detection system: A survey. *Journal of Network and Computer Applications*, 68:90–113.
- [Aggarwal, 2017] Aggarwal, C. C. (2017). *Outlier Analysis*. Springer International Publishing.
- [Chawla et al., 2002] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357.
- [Elkan, 2001] Elkan, C. (2001). The foundations of cost-sensitive learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, pages 973–978, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- [Fernández et al., 2018] Fernández, A., García, S., Herrera, F., and Chawla, N. V. (2018). Smote for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary. *Journal of Artificial Intelligence Research*, 61:863–905.
- [Tomek, 1976] Tomek, I. (1976). Two modifications of cnn. *IEEE Trans. Systems, Man and Cybernetics*, 6:769–772.
- [Wilson, 1972] Wilson, D. L. (1972). Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-2(3):408–421.