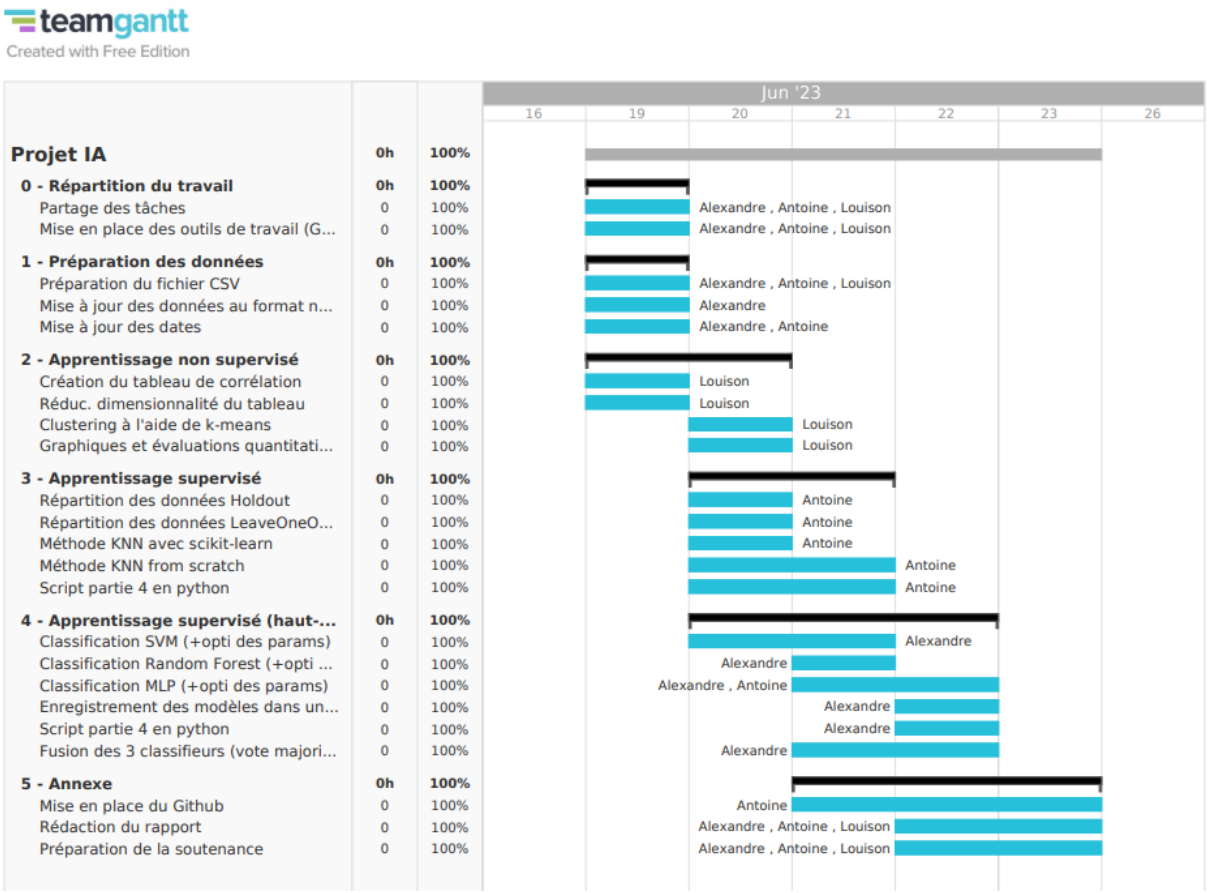
DIGUER Louison
RADIN Alexandre
SOYDEMIR Antoine
CIR3

PROJET IA A3

KARINE Ayoub et ABOU EL HOUDA Zakaria
23/06/2023



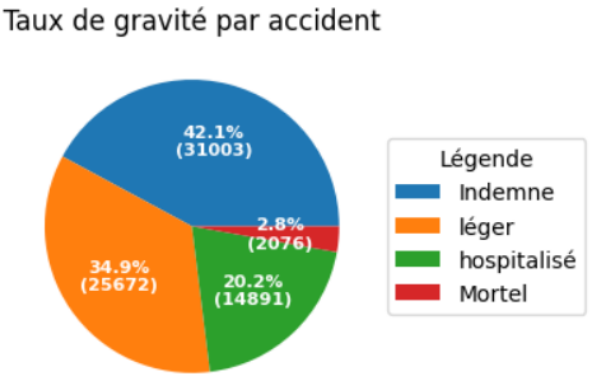
1 - Découverte et préparation des données

1.1 - Préparation de l'environnement de travail et gestion de projet

1.2 - Découverte des données

Notre base de données comporte 22 colonnes (features) et 73643 lignes (instances). La gravité de l'accident possède 4 classes:

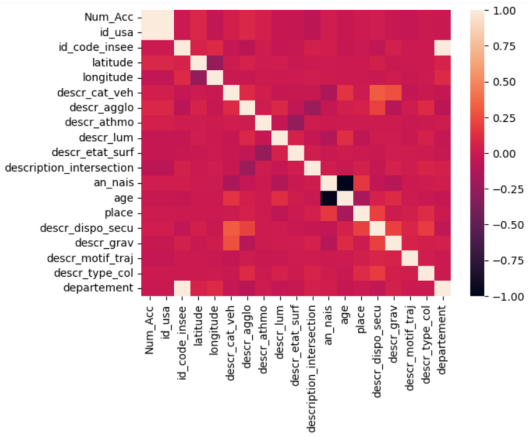
- 1 = blessé hospitalisé (31 004 instances)
- 2 = blessé léger (25 672 instances)
- 3 = indemne (14 891 instances)
- 4 = mortel (2 076 instances)



2 - Apprentissage non-supervisé

2.1 - Réduction de dimension

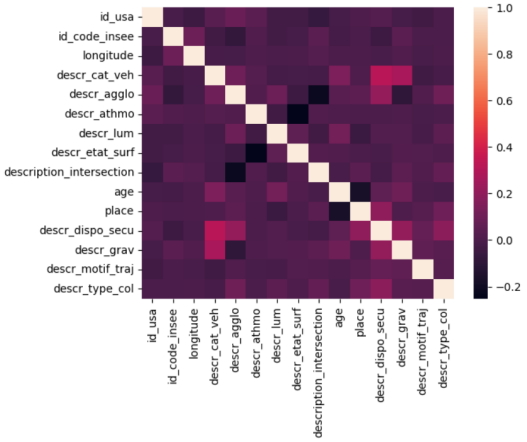
Pour ce qui est de l'apprentissage non-supervisé, nous avons commencé par chercher la corrélation entre les différents attributs afin de supprimer ceux qui étaient très corrélés (coefficient de corrélation supérieur à 0.7 ou inférieur à -0.7). Ainsi, nous pourrions supprimer l'un des attributs corrélés afin de diminuer la dimension de notre data frame.



A l'aide de la librairie seaborn, nous avons tracé la heatmap de la matrice de corrélation de toutes nos features. Nous remarquons ainsi que les données corrélées sont :

- Num_Acc/id_usa
- id_code_insee/departement
- latitude/longitude
- an_nais/age

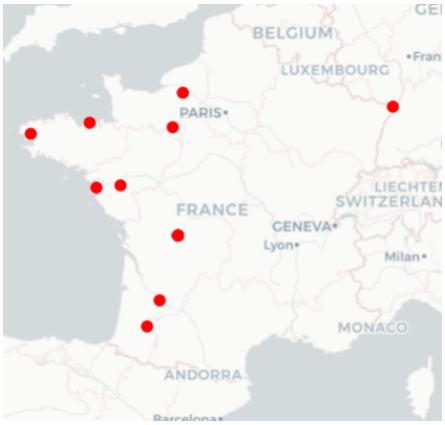
Après suppression de certaines features (features en rouge), nous trouvons la heatmap de corrélation suivante. Ainsi, nous avons supprimé 4 des 21 features de notre data frame ce qui représente un pourcentage de réduction de 18.18%.



2.2 - Partitionnement (clustering)

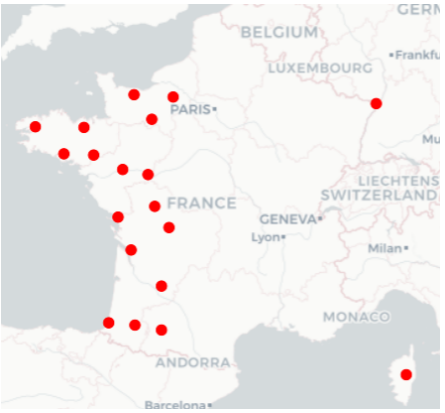
Pour cette partie, nous avons fait de la programmation orientée objet en créant une classe KMeans_from_scratch qui possède les mêmes attributs et méthodes clés que la classe KMeans de la librairie scikit-learning (sklearn).

Notre objet KMeans_from_scratch effectue une initialisation aléatoire des centroïdes pour les clusters. Des indices aléatoires sont générés à partir des données d'entrée afin de sélectionner les points initiaux pour les centroïdes. Ensuite, l'algorithme itère sur un maximum d'itérations définies. À chaque itération, les points sont attribués aux centroïdes les plus proches. Il calcule la distance entre chaque point et les centroïdes. La distance utilisée peut être euclidean, manhattan ou haversine. Après l'attribution des points aux clusters, les centroïdes sont mis à jour. Les nouveaux centroïdes sont calculés en prenant la moyenne des points appartenant à chaque cluster. Cette étape de mise à jour est répétée jusqu'à ce que le nombre maximum d'itérations soit atteint.



A l'aide de la librairie graph_object de plotly, nous avons tracé différentes cartes selon la méthode de calcul de distance utilisée ainsi que l'objet KMeans utilisé. La carte ci-contre représente les centroïdes (clusters) qui ont été prédit par notre classe KMeans_from_scratch avec le calcul de distance euclidien car il s'agit de la même méthode de calcul de distance que le KMeans de chez sklearn. Notre modèle n'est pas très précis car certains clusters sont superposés.

L'autre carte représente les clusters prédit par l'objet KMeans de chez sklearn. Nous pouvons remarquer que leur modèle est beaucoup plus précis que le nôtre car il n'y a aucun cluster qui est superposé. De plus, nous voyons qu'un cluster a été placé en Corse par leur modèle ce qui n'est pas le cas de notre modèle.



2.3 - Étude qualitative des résultats

Afin d'étudier les résultats obtenus grâce à nos deux modèles, nous avons utilisé 3 méthodes. La première étant le score silhouette, qui est la différence entre la distance moyenne d'un point avec les points du même cluster et la distance moyenne du point avec les points des autres clusters. Les résultats de ce score vont de -1 (mauvaise classification) à 1 (bonne classification). Avec différents nombre de clusters, nous avons fait des tests et obtenus les résultats suivants:

KMeans\cluster	5	10	15	20	25	30	35	40	45	50
FS euclidian	0.524034	0.496133	0.648167	0.585722	0.570483	0.608618	0.658289	0.63923	0.673448	0.638631
FS manhattan	0.567909	0.616585	0.583598	0.616976	0.579025	0.633244	0.621491	0.714025	0.69473	0.683205
FS haversine	0.409283	0.493031	0.47644	0.493608	0.531936	0.527755	0.540506	0.576528	0.529105	0.609056
sklearn	0.680573	0.72884	0.731803	0.746002	0.771416	0.78369	0.799725	0.81814	0.826161	0.842601

Grâce à ce tableau, nous pouvons remarquer que le modèle de chez sklearn est celui qui obtient le meilleur score en tout point. Le deuxième meilleur est le modèle KMeans from scratch avec le calcul de distance manhattan.

Le deuxième score utilisé est l'indice Calinski-Harabasz. Le calcul de ce score est basé sur le rapport entre la variance inter-groupe et la variance intra-groupe. Les résultats de ce score peuvent aller de 0 (mauvaise classification) à +infini (bonne classification). Les résultats obtenus sont les suivants:

KMeans\cluster	5	10	15	20	25	30	35	40	45	50
FS euclidian	43746.5	32491.5	34727.7	81319.1	57098.6	22536	68888.6	25083.2	69264.3	66530.8
FS manhattan	45750.1	42321.3	26013.7	64759.4	51345.9	22029.7	17418.1	60286.9	53748.7	17534.3
FS haversine	59245.6	35985.4	20060.4	27752.5	23227.3	19948.9	22771.1	18061.2	16023.8	49990.2
sklearn	101116	182980	236614	281621	350178	390982	444331	504735	562744	623050

On conclut de ce tableau que la meilleure méthode reste celle de sklearn qui a un score 2 fois plus important que le score des autres méthodes. Cela est dû au fait que ce score est utilisé par sklearn afin de vérifier la convergence des points. Alors que notre classe from scratch s'arrête en fonction du nombre d'itération maximum demandé.

Le troisième et dernier score utilisé est l'indice de Davies-Bouldin. Ce score est calculé grâce à la moyenne du rapport maximal entre la distance d'un point au centre de son groupe et la distance entre deux centres de groupes. Les résultats de ce score peuvent aller de 0 (bonne classification) à +infini (mauvaise classification). Il faut toujours comparer ce score à la taille de l'échantillon (ici, plus de 73 000). Les résultats obtenus sont les suivants:

KMeans\cluster	5	10	15	20	25	30	35	40	45	50
FS euclidian	0.774011	0.811766	0.743962	0.953988	1.00938	0.977539	0.831136	0.821504	1.0768	1.31658
FS manhattan	0.92844	0.990488	1.27506	1.0693	0.835342	1.01699	0.651721	1.02586	0.900541	0.913095
FS haversine	0.772077	1.02349	0.871202	0.882489	0.888187	1.00182	1.19488	1.30921	0.947916	0.896864
sklearn	0.545981	0.47464	0.581558	0.555695	0.55238	0.579249	0.52346	0.549306	0.452548	0.461527

Une fois de plus, nous pouvons remarquer que la classe sklearn est celle qui obtient les meilleurs scores. Nous remarquons aussi que les modèles KMeans from scratch obtiennent tout de même de très bon score du fait que ce soit inférieur ou autour de 1.

Pour conclure, nous pouvons dire que le modèle KMeans de chez sklearn autant au niveau des scores qu'au niveau de la complexité de l'algorithme. En effet, tous les tableaux nous montrent que KMeans sklearn obtient de très bons scores peu importe le calcul de score choisi. Au niveau de la complexité, nous avons pu constater que le modèle de chez sklearn mettait moins de temps à donner un résultat que notre modèle from scratch. Nous avons pu aussi constater que notre modèle from scratch obtient tout de même de bons scores (hormis avec l'indice de Calinski-Harabasz), ce qui nous permet de dire que nous avons un modèle qui fonctionne correctement. Nous pouvons expliquer la différence de score du fait que le

modèle de chez sklearn utilise deux conditions de fin qui sont soit la variable max_iter, soit un calcul par différents scores dont l'indice Calinski-Harabasz. Tandis que notre modèle possède comme seule condition de fin la variable max_iter.

3 - Apprentissage supervisé

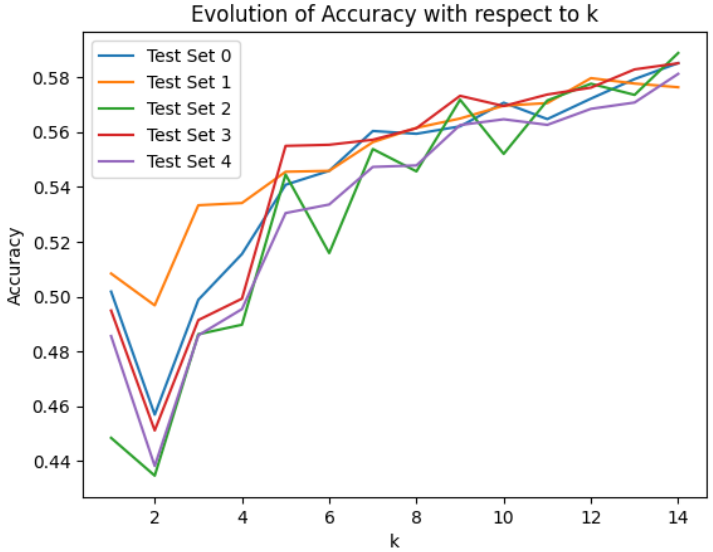
3.1 - Répartition des données

Pour la répartition des données, nous avons utilisé 2 algorithmes "Holdout" et "Leave-one-out". Pour le premier algorithme "Holdout" il s'agit de répartir aléatoirement les données. L'avantage de cet algorithme est qu'il permet de fournir de manière impartiale les performances du modèle sur des données nouvelles (ensemble de test). Néanmoins, les performances du modèle peuvent varier car elles dépendent de comment les données ont été divisés. C'est pour cela que dans le cas de notre étude, on réalisera successivement 5 fois la méthode Holdout pour observer et prendre en compte les possibles écarts créés par le facteur aléatoire de cette méthode.

Pour le deuxième algorithme "Leave-one-out", cet algorithme consiste à itérer sur chaque échantillon des données. A chaque itération, il va exclure un échantillon et le considéré comme étant la base de test, tandis que le reste des données sera utilisé comme base d'apprentissage. On parle notamment de validation croisée dans ce cas. Ce deuxième algorithme est beaucoup plus coûteux en termes de ressources. Pour ces raisons, nous nous baserons sur un échantillon de 10% de la base de données pour effectuer nos tests.

3.2 - Classification avec KNN

Après avoir préparé les données, on exécute l'algorithme KNN (K-Nearest Neighbors). Pour utiliser cet algorithme de classification il est nécessaire de choisir une valeur de k. Dans le cas de notre étude, nous avons fait varier cette valeur de 1 à 15 pour observer les différents résultats que l'on peut obtenir. L'algorithme va ensuite calculer la distance entre le nouvel exemple qui doit être classifier et les données d'entraînements. Cette distance peut-être calculée de différente manière mais la plus classique est la distance euclidienne. Dans le cas de notre étude, nous avons également traiter manhattan et minkowski. Une fois les distances calculées, l'algorithme va choisir les k plus proches voisins pour l'exemple traité.



La classe du nouvel exemple est choisie à travers un vote majoritaire qui sélectionne la classe la plus fréquente parmi les k plus proches voisins.

A l'aide la librairie sklearn et d'une répartition des données avec l'algorithme Holdout on obtient en moyenne une précision de 54% avec ce modèle. De plus, on peut constater l'effet aléatoire de la méthode Holdout, qui se traduit par des écarts entre les différents ensembles de données créés à partir de cette méthode.

Avec la méthode Leave-one-out on obtient une moyenne des performances de 51% en prenant comme données 10% de l'ensemble des données d'entrées et 15 comme valeur de k.

On peut donc conclure que la méthode Holdout est plus adaptée dans le cas de notre étude, car elle offre une meilleure précision et une meilleure généralisation des données.

La méthode "from scratch" à été développée avec différentes distances : euclidienne, manhattan et minkwoski.

```
Metric: euclidean, accuracy: 53.768 %
Metric: manhattan, accuracy: 53.734 %
Metric: minkowski, accuracy: 53.768 %
```

Ces tests ont été réalisés sur un échantillon de 20% de la base de données. En effet, la méthode from scratch étant moins optimisée consomme des ressources importantes notamment à l'étape de calculs des distances. Le temps d'exécution étant chronophage, nous avons fait le choix d'exécuter cette méthode sur un échantillon. Dans le cas de cet exemple nous avons choisi k = 5. Bien que les ressources nécessaires à l'exécution soient importantes, on parvient à obtenir une précision proche (0.4% près) de celle obtenue par le biais de la librairie sklearn. On observe des écarts très faibles entre les différentes distances, mais la distance euclidienne semble être la référence parmi les différentes méthodes.

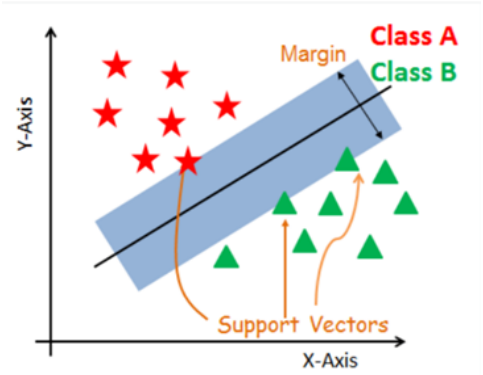
3.3 - Classification avec trois algorithmes de "haut-niveau"

Support Vector Machine (SVM)

Le SVM est un modèle de classification qui a besoin de beaucoup de temps d'exécution. Selon la documentation sklearn associée, « The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples ». Dans notre cas, le dataset utilisé est très grand ; nous allons donc extraire un échantillon de la base de données pour entraîner le modèle, puisque le SVM est plus approprié pour les petits datasets.

Le concept de l'algorithme est de séparer les points des différentes classes en utilisant des hyperplans dans des espaces multi-dimensionnels. Le but est aussi d'avoir des marges larges autour des hyperplans. L'algorithme génère donc un hyperplan optimal d'une manière

itérative. Le but est de trouver le maximum marginal hyperplane (MMH) qui divise le mieux le dataset en classe, afin de classer de nouveaux points.



[Scikit-learn SVM Tutorial with Python (Support Vector Machines) | DataCamp]

Les support vectors sont les data points qui sont le plus proches de l'hyperplan. Ils vont servir à définir la ligne de séparation en calculant les marges.

Une bonne marge est une marge avec beaucoup d'espaces entre les classes. Donc, on veut trouver la meilleure marge.

Cependant, des problèmes ne peuvent pas être résolus en utilisant des hyperplans linéaires. L'algorithme va donc utiliser des astuces de noyau pour transformer l'espace d'entrée en un espace de dimension supérieure afin de convertir des problèmes non séparables en des problèmes séparables.

Dans notre étude, on va donc essayer de trouver les hyperparamètres optimaux du modèle. Pour ce faire, on va ajuster trois hyperparamètres principaux :

1. Kernels : les noyaux ont pour but de transformer la dimension des données.
2. C : c'est le paramètre de pénalité représentant une mauvaise classification ou un terme d'erreur. Plus C est grand, plus la classification va être correcte, mais on a un risque d'overfit. Une petite valeur de C créée un hyperplan avec des petites marges alors qu'une grande valeur de C créé un hyperplan avec des grandes marges.
3. Gamma : il définit l'influence de la ligne de séparation. Quand Gamma est grand, les points proches sont considérés et ont une grande influence. Quand il est petit, les points distants sont aussi considérés pour obtenir la limite de décision.

Dans un premier temps, on entraîne le modèle sur les données d'entraînement avec les paramètres de base. On effectue ensuite une prédiction sur les données de test. On affiche l'accuracy score, qui est le nombre de prédictions correctes divisé par le nombre total de prédictions, à partir des données prédites et des données réelles. On obtient :

```
SVC accuracy score  (without hyper parameter tuning) : 0.40727667662231876
```

Ce score est assez bas. On affiche donc aussi la précision, le rappel et le f1-score du modèle :

	precision	recall	f1-score	support
1	0.00	0.00	0.00	749
2	0.00	0.00	0.00	1317
3	0.41	1.00	0.58	1500
4	0.00	0.00	0.00	117
accuracy			0.41	3683
macro avg	0.10	0.25	0.14	3683
weighted avg	0.17	0.41	0.24	3683

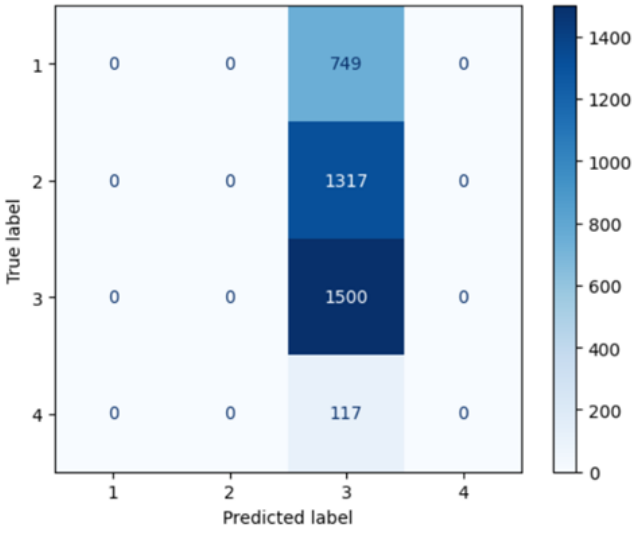
Le rappel permet de savoir le pourcentage de positifs bien prédits par le modèle. C'est le nombre de vrais positifs (positifs bien prédits) sur l'ensemble des positifs (vrais positifs + faux négatifs). Plus il est élevé, plus on maximise le nombre de vrais positifs.

La précision permet de savoir le nombre de prédictions positives bien effectuées. C'est le nombre de positifs bien prédits (vrais positifs) sur l'ensemble des positifs prédits (vrais positifs + faux positifs). Plus elle est élevée, plus le modèle minimise le nombre de faux positifs.

$$F1\ Score = 2 \times \frac{recall \times precision}{recall + precision}$$

Mais, ces métriques ne sont pas très utiles séparément. On introduit donc le f1-score qui combine les deux. Le f1-score évalue la performance du modèle.

Plus il est élevé, plus le modèle est performant. On affiche aussi la matrice de confusion :



Elle montre les labels réels par rapport aux labels prédits. On voit que le modèle ne prédit que des classes 3. Toutes les classes 3 réelles sont bien prédites. C'est pour cela qu'on a un rappel de 1. Mais avec le f1-score, on voit que le modèle n'est pas performant.

On cherche donc les paramètres optimaux du modèle et on trouve :

```
SVC : Best parameters : {'C': 1, 'gamma': 0.01}
SVC : Best estimator : SVC(C=1, gamma=0.01)
```

En faisant une nouvelle prédiction avec ces paramètres, on a :

```
SVC accuracy score  (hyper parameter tuning) : 0.41542221015476516
```

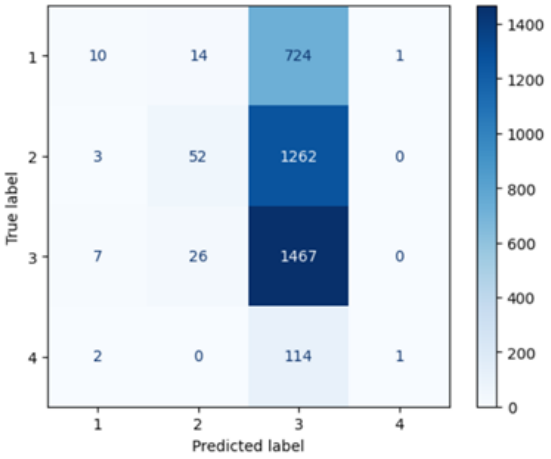
On a les métriques :

	precision	recall	f1-score	support
1	0.45	0.01	0.03	749
2	0.57	0.04	0.07	1317
3	0.41	0.98	0.58	1500
4	0.50	0.01	0.02	117
accuracy			0.42	3683
macro avg	0.48	0.26	0.17	3683
weighted avg	0.48	0.42	0.27	3683

On a donc une amélioration de la précision : le modèle minimise le nombre de faux positifs. Par exemple, si une classe est prédite comme 1, il

y a plus de chances que ce soit effectivement le cas.

Par exemple, en affichant la matrice de confusion, sur les 10 labels 1 prédits, on a 45% de chance que ce soit effectivement un 1.



On souhaite avoir la majorité des valeurs dans la diagonale (labels prédits = labels réels). On voit que le modèle se trompe moins qu'avant, mais les bonnes prédictions effectuées sont négligeables au regard du nombre d'occurrences de chaque classe. Le modèle n'est donc pas adapté.
On a peut-être ici un problème d'overfitting : le modèle apprend peut-être très bien dans la base d'entraînement mais n'arrive pas à généraliser sur de nouvelles valeurs.

NB : Dans les paramètres optimaux, les différentes valeurs du kernel ont été testées. Il ressort que "linear" et "poly" demandent un temps d'exécution beaucoup trop long. C'est pourquoi dans la recherche des paramètres optimaux, le paramètre kernel n'a pas été testé (on a gardé "rbf" par défaut).

Random Forest

Cet algorithme repose sur des arbres de décision. Un arbre de décision sert à prendre une décision grâce à une série de questions dont la réponse va mener à la décision finale.

Tous les arbres de décision composant la forêt sont entraînés avec un sous-ensemble de données du problème. Ces données sont choisies aléatoirement afin que certains arbres y aient accès et d'autres non pour que chaque arbre ait une expérience différente du problème. Quand tous les arbres sont entraînés, le Random Forest prend la décision finale en faisant voter tous les arbres de décision le composant. La majorité l'emporte.

Il y a donc plusieurs hyperparamètres qui composent ce modèle. Par exemple, en affinant l'hyperparamètre n_estimator, on peut choisir le nombre d'arbres qui composent la forêt.

Donc, lorsque l'on construit un modèle de classification, on doit savoir quelles valeurs donner aux hyperparamètres. Dans notre cas, on va régler différents hyperparamètres :

1. **n_estimator** : définit le nombre d'arbres
2. **max_depth :** définit le nombre maximum de divisions de chaque arbre. Ce paramètre est donc important pour éviter le surapprentissage.

Avec le modèle de base, on a :

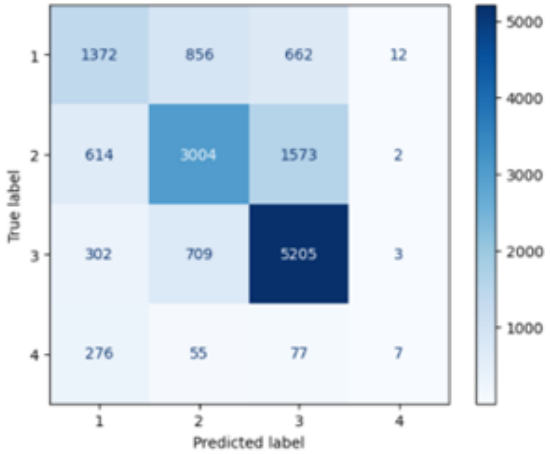
```
Random Forest accuracy score  (without hyper parameter tuning) : 0.6509606897956413
```

On a aussi les métriques suivantes :

	precision	recall	f1-score	support
1	0.54	0.47	0.50	2902
2	0.65	0.58	0.61	5193
3	0.69	0.84	0.76	6219
4	0.29	0.02	0.03	415
accuracy			0.65	14729
macro avg	0.54	0.48	0.48	14729
weighted avg	0.64	0.65	0.64	14729

En regardant les f1-scores, on remarque que le modèle n'est pas performant sur les classes 4.

C'est confirmé par la matrice de confusion :



La classe 4 est prédite en majorité comme étant la classe 1.
Ce modèle est cependant plus performant que le précédent avec de bonnes valeurs dans les diagonales.

On améliore le modèle en testant différentes combinaisons de paramètres. On obtient :

```
Position : 1 - Score : 0.649302 (+/-0.002055) for {'max_depth': 20, 'n_estimators': 500}

Random Forest : Best parameters : {'max_depth': 20, 'n_estimators': 500}
Random Forest : Best estimator : RandomForestClassifier(max_depth=20, n_estimators=500)
```

Sur les différentes combinaisons testées, le score moyen des 5 folds est donc le meilleur pour la combinaison max_depth = 20 et n_estimators = 500.

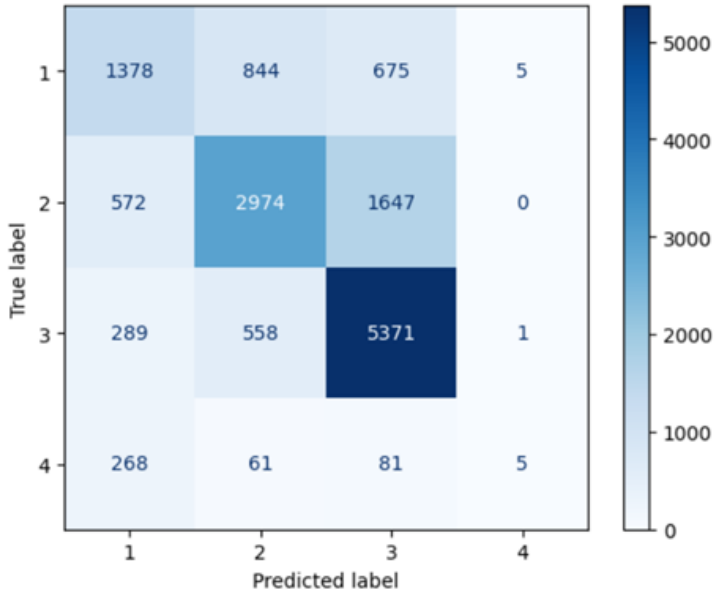
En entraînant le modèle avec ces valeurs, on obtient les résultats suivants :

```
Random Forest accuracy score  (hyper parameter tuning) : 0.6604657478443886
```

	precision	recall	f1-score	support
1	0.55	0.47	0.51	2902
2	0.67	0.57	0.62	5193
3	0.69	0.86	0.77	6219
4	0.45	0.01	0.02	415
accuracy			0.66	14729
macro avg	0.59	0.48	0.48	14729
weighted avg	0.65	0.66	0.64	14729

La précision associée à la classe 4 est plus élevée. Le modèle minimise le nombre de faux positifs.

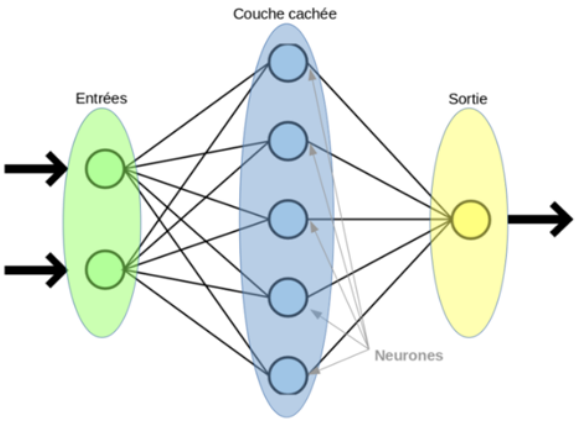
La matrice de confusion est la suivante :



On a ici la valeur 5 au croisement 4-4, ce qui veut peut-être dire que le modèle d'avant avait prédit deux 4 de manière erronée (faux positifs).

Multilayer Perceptron (MLP)

Le MLP (perceptron multicouche) est un type de réseau de neurones. Il est donc composé de neurones reliés entre eux par des connexions. On associe un poids à chaque connexion (entre 0 et 1). Les neurones sont organisés en couches (une couche d'entrée, une de sortie, et une ou plusieurs couches entre les deux appelées couches cachées).



[Fonctionnement du perceptron multicouche – Bloom Magazine (home.blog)]

On va donc ajuster plusieurs paramètres pour trouver le modèle optimal.

1. **hidden_layer_sizes** : nombres de neurones dans les différentes couches cachées
2. **activation** : fonction pour les couches cachées
3. **solver** : le solver utilisé pour optimiser les poids
4. **alpha**
5. **learning rate**

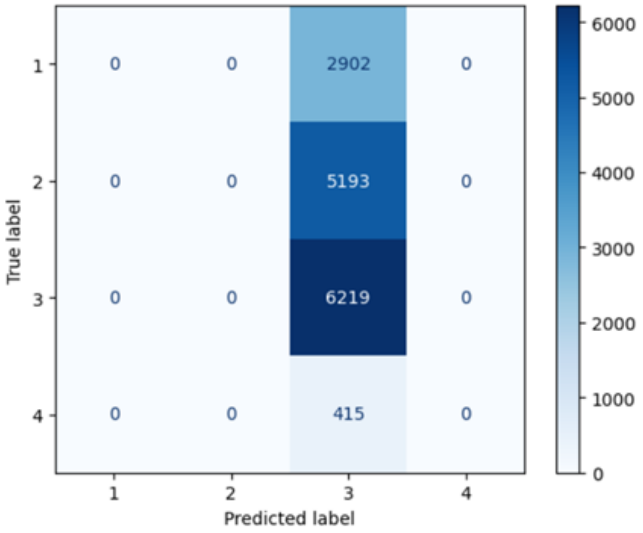
On a le modèle de base suivant :

```
MLP accuracy score  (without hyper parameter tuning) : 0.4222282571797135
```

	precision	recall	f1-score	support
1	0.00	0.00	0.00	2902
2	0.00	0.00	0.00	5193
3	0.42	1.00	0.59	6219
4	0.00	0.00	0.00	415
accuracy			0.42	14729
macro avg	0.11	0.25	0.15	14729
weighted avg	0.18	0.42	0.25	14729

Le modèle semble encore une fois défaillant.

On affiche la matrice de confusion :



Toutes les classes sont prédites comme appartenant à la classe 3.

En cherchant les paramètres optimaux, on trouve :

```
MLP : Best parameters : {'activation': 'tanh', 'alpha': 0.05, 'hidden_layer_sizes': (100,), 'learning_rate': 'constant', 'solver': 'adam'}
MLP : Best estimator : MLPClassifier(activation='tanh', alpha=0.05)
```

Le score, les métriques (rappel, précision, f1-score) et la matrice de confusion sont exactement les mêmes que le modèle de base.

Une hypothèse de la faible performance des modèles de classification SVM et MLP vient peut-être du fait du fléau de la dimension. C'est lorsque l'on a un risque d'overfitting lié au surplus de dimensions.

Fusion des trois classifieurs (Vote majoritaire)

Un Voting Classifier est un modèle de machine learning qui s'entraîne sur un ensemble de modèles et qui prédit des classes basées sur la probabilité la plus élevée qu'elles soient

choisies. L'algorithme agrège les résultats de chaque classificateur donné et prédit la classe de sortie en fonction de la plus grande majorité des votes.

On a deux types de vote :

1. Hard voting : la classe prédite est la classe avec la plus grande majorité de votes, c'est-à-dire la classe qui a la plus forte probabilité d'être prédite par chacun des classifieurs. Si les trois classifieurs prédisent la classe (A, A, B), alors la majorité prédit A. A est donc la prédiction finale.

2. Soft voting : la classe prédite est basée sur la moyenne des probabilités donnée à cette classe. Si on donne des entrées à nos trois modèles et qu'on obtient la probabilité de prédiction pour la classe A = (0.3, 0.47, 0.53) et B = (0.2, 0.32, 0.4), alors la moyenne pour la classe A est 0.433 et B est 0.3067. C'est donc A qui est la prédiction finale.

Voting Classifier ne marche pas sur des modèles déjà entraînés selon la documentation : « Soft Voting/Majority Rule classifier for unfitted estimators. » Donc, on va passer en argument de Voting Classifier les modèles avec les paramètres optimaux pour qu'il puisse les entraîner.

On obtient le résultat suivant pour le hard voting :

```
Hard Voting Score : 0.44354674451761833
```

Ce score est obtenu en faisant une prédiction à partir du Voting Classifier entraîné avec les trois modèles. Le score n'est pas élevé puisque le modèle SVM et le modèle MLP ne sont pas performants. Ils sont majoritaires sur Random Forest, et donc la décision finale est prise à l'encontre de Random Forest.

Pour le soft voting, on a :

```
Soft Voting Score : 0.6318826804263697
```

Cette différence est obtenue grâce à un paramètre supplémentaire ajouté au SVC (obligatoire dans le cas d'un soft voting pour pouvoir prédire de nouvelles valeurs par la suite). Son entraînement diffère donc du précédent, ce qui peut expliquer cette différence.