

## Test de SMOTE et ADASYN sur des Données Déséquilibrées

Dans ce notebook, nous allons :

- Générer un dataset artificiellement déséquilibré, puis appliquer SMOTE et ADASYN pour rééquilibrer les classes.
- Tester les mêmes approches sur le dataset Titanic ainsi que sur un sous-ensemble déséquilibré de MNIST.
- Évaluer les performances des modèles (Random Forest / Logistic Regression) avant et après rééchantillonnage.
- Visualiser les distributions de classes, les courbes ROC, les matrices de confusion (sous forme de heatmap), etc.
- Discuter des constatations, des performances observées et des moyens d'amélioration (en jouant notamment sur les paramètres `k_neighbors` pour SMOTE/ADASYN et `n_estimators` pour le RandomForest).

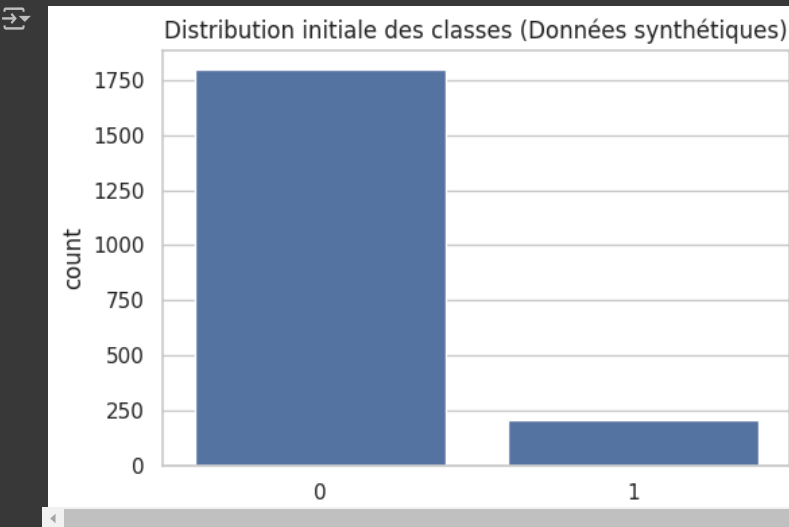
```
1 #!pip install imbalanced-learn
2
3 import numpy as np
4 import pandas as pd
5 import matplotlib.pyplot as plt
6 import seaborn as sns
7
8 from sklearn.datasets import make_classification
9 from sklearn.model_selection import train_test_split, GridSearchCV
10 from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
11 from sklearn.ensemble import RandomForestClassifier
12 from sklearn.preprocessing import StandardScaler
13
14 from imblearn.over_sampling import SMOTE, ADASYN
15 from imblearn.pipeline import Pipeline
16
17 from tensorflow.keras.datasets import mnist
18
19 sns.set(style="whitegrid")
20
```

### Données Synthétiques

Objectif : Créer un jeu de données artificiel déséquilibré, puis comparer les performances d’un modèle avant et après application de SMOTE et ADASYN.

Données : Nous utilisons `make_classification` pour créer un dataset avec 2000 échantillons, où environ 10% des données appartiennent à la classe minoritaire.

```
1 X, y = make_classification(n_samples=2000,
2                           n_features=2,
3                           n_informative=2,
4                           n_redundant=0,
5                           n_clusters_per_class=1,
6                           weights=[0.9, 0.1],
7                           class_sep=1.0,
8                           random_state=42)
9
10 plt.figure(figsize=(6,4))
11 sns.countplot(x=y)
12 plt.title("Distribution initiale des classes (Données synthétiques)")
13 plt.show()
14
```



### Modèle Sans Rééchantillonnage (Données Synthétiques)

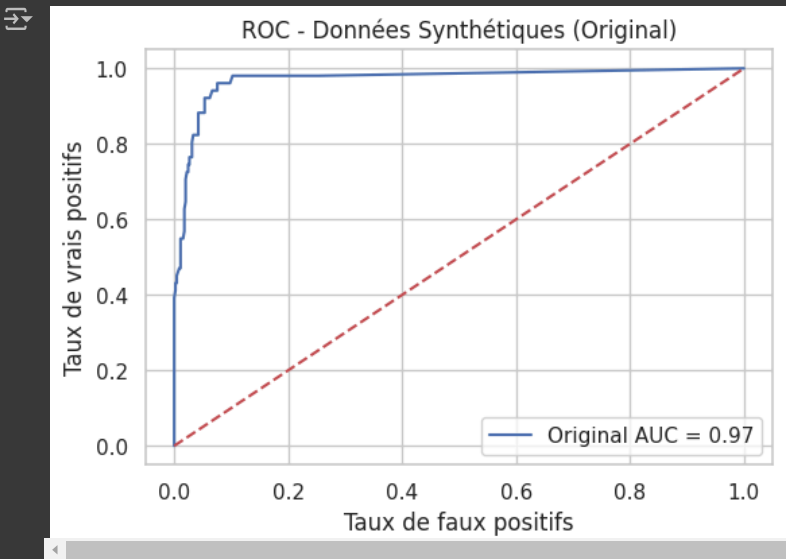
```
1 X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
2
3 scaler = StandardScaler()
4 X_train_scaled = scaler.fit_transform(X_train)
5 X_test_scaled = scaler.transform(X_test)
6
7 # Modèle sans rééchantillonnage
```

```
8 model = RandomForestClassifier(random_state=42)
9 model.fit(X_train_scaled, y_train)
10 y_pred = model.predict(X_test_scaled)
11
12 print("Performances sans rééchantillonnage:")
13 print(classification_report(y_test, y_pred))
14
```

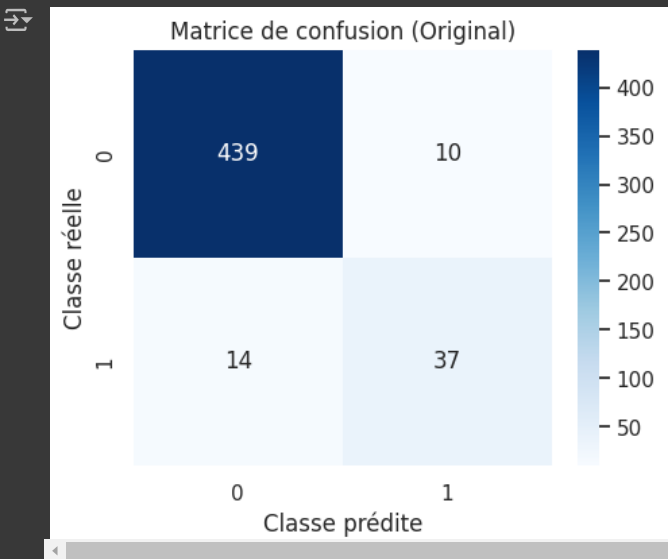
↗

| Performances sans rééchantillonnage: |           |        |          |         |  |
|--------------------------------------|-----------|--------|----------|---------|--|
|                                      | precision | recall | f1-score | support |  |
| 0                                    | 0.97      | 0.98   | 0.97     | 449     |  |
| 1                                    | 0.79      | 0.73   | 0.76     | 51      |  |
| accuracy                             |           |        | 0.95     | 500     |  |
| macro avg                            | 0.88      | 0.85   | 0.86     | 500     |  |
| weighted avg                         | 0.95      | 0.95   | 0.95     | 500     |  |

```
1 fpr, tpr, thresholds = roc_curve(y_test, model.predict_proba(X_test_scaled)[: ,1])
2 auc_score = auc(fpr, tpr)
3
4 plt.figure(figsize=(6,4))
5 plt.plot(fpr, tpr, label='Original AUC = %.2f' % auc_score)
6 plt.plot([0,1],[0,1], 'r--')
7 plt.title("ROC - Données Synthétiques (Original)")
8 plt.xlabel("Taux de faux positifs")
9 plt.ylabel("Taux de vrais positifs")
10 plt.legend()
11 plt.show()
12
```



```
1 cm = confusion_matrix(y_test, y_pred)
2 plt.figure(figsize=(5,4))
3 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
4 plt.title("Matrice de confusion (Original)")
5 plt.ylabel("Classe réelle")
6 plt.xlabel("Classe prédite")
7 plt.show()
8
```



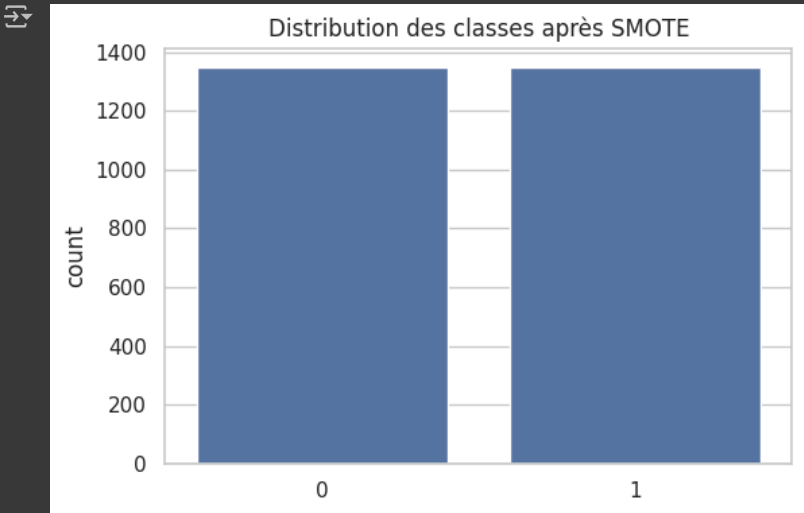
Interprétation :

Le modèle montre une bonne performance globale, comme indiqué par une AUC de 0.97 sur la courbe ROC. Cela traduit une capacité élevée du modèle à distinguer les classes.

Cependant, l'analyse de la matrice de confusion révèle que, bien que le modèle performe bien sur la classe majoritaire (classe 0 avec 439 vraies prédictions positives), il montre une performance relativement faible sur la classe minoritaire (classe 1), avec seulement 37 instances correctement prédites. Ce déséquilibre de performance se reflète dans un rappel plus faible pour la classe minoritaire.

En conclusion, bien que le modèle soit précis dans son ensemble, il présente un biais vers la classe majoritaire.

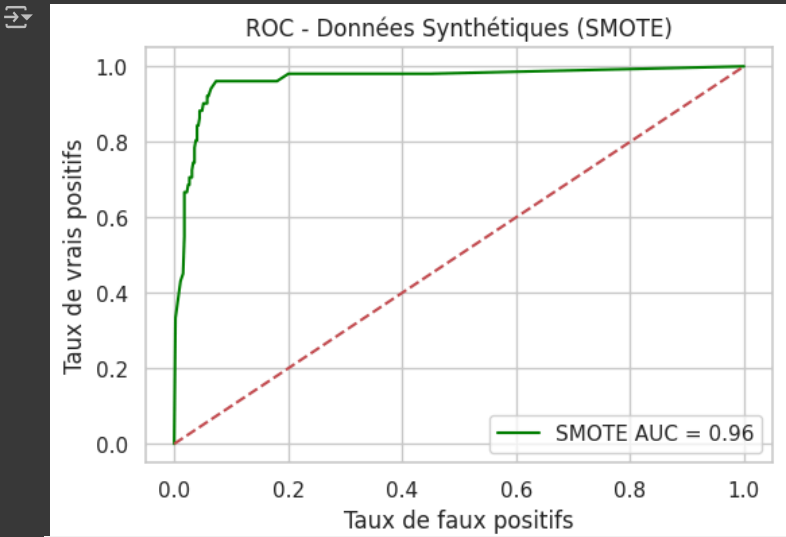
```
1 sm = SMOTE(random_state=42)
2 X_train_sm, y_train_sm = sm.fit_resample(X_train_scaled, y_train)
3
4 # Distribution après SMOTE
5 plt.figure(figsize=(6,4))
6 sns.countplot(x=y_train_sm)
7 plt.title("Distribution des classes après SMOTE")
8 plt.show()
9
10 model_sm = RandomForestClassifier(random_state=42)
11 model_sm.fit(X_train_sm, y_train_sm)
12 y_pred_sm = model_sm.predict(X_test_scaled)
13
14 print("Performances après SMOTE:")
15 print(classification_report(y_test, y_pred_sm))
16
```



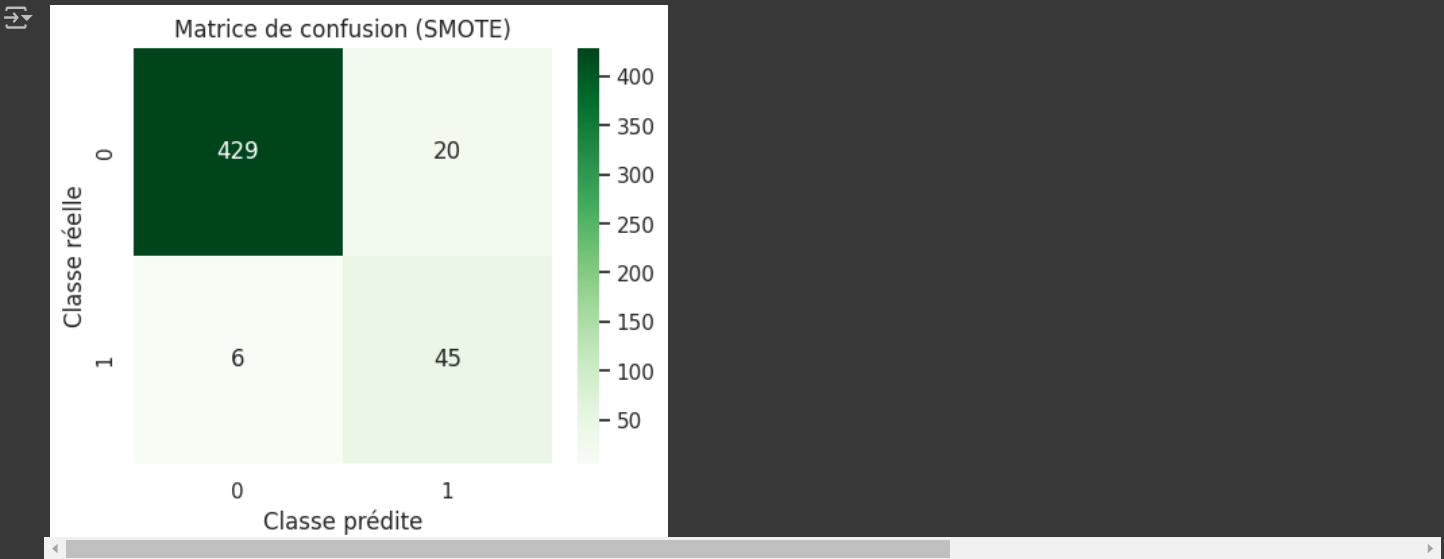
Performances après SMOTE:

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.99      | 0.96   | 0.97     | 449     |
| 1            | 0.69      | 0.88   | 0.78     | 51      |
| accuracy     |           |        | 0.95     | 500     |
| macro avg    | 0.84      | 0.92   | 0.87     | 500     |
| weighted avg | 0.96      | 0.95   | 0.95     | 500     |

```
1 fpr_sm, tpr_sm, _ = roc_curve(y_test, model_sm.predict_proba(X_test_scaled)[: ,1])
2 auc_sm = auc(fpr_sm, tpr_sm)
3
4 plt.figure(figsize=(6,4))
5 plt.plot(fpr_sm, tpr_sm, label='SMOTE AUC = %.2f' % auc_sm, color='green')
6 plt.plot([0,1],[0,1], 'r--')
7 plt.title("ROC - Données Synthétiques (SMOTE)")
8 plt.xlabel("Taux de faux positifs")
9 plt.ylabel("Taux de vrais positifs")
10 plt.legend()
11 plt.show()
12
```



```
1 cm_sm = confusion_matrix(y_test, y_pred_sm)
2 plt.figure(figsize=(5,4))
3 sns.heatmap(cm_sm, annot=True, fmt='d', cmap='Greens')
4 plt.title("Matrice de confusion (SMOTE)")
5 plt.ylabel("Classe réelle")
6 plt.xlabel("Classe prédite")
7 plt.show()
8
```



Interprétation :

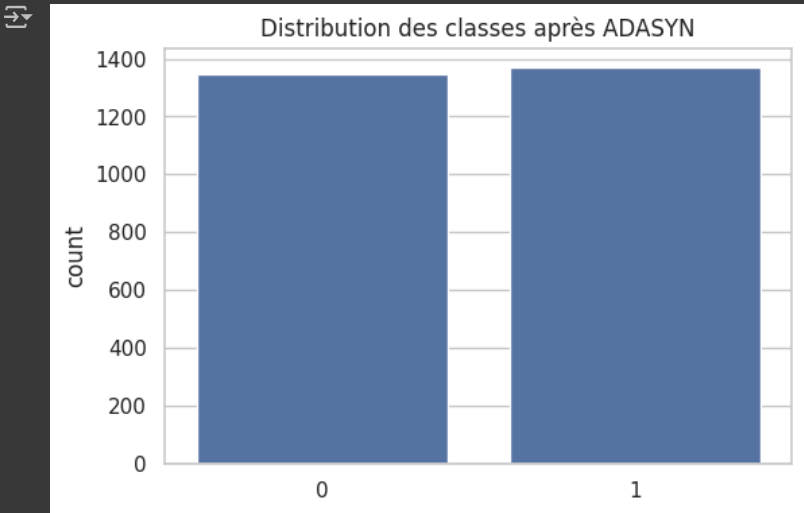
L'utilisation de SMOTE pour générer des exemples synthétiques de la classe minoritaire a permis d'améliorer la détection de cette dernière. Cela se reflète dans une augmentation du rappel pour la classe 1, avec 45 instances correctement classées contre 37 précédemment. Cette amélioration est également visible dans la réduction du nombre de faux négatifs (6 contre 14 dans le cas initial).

Cependant, cette amélioration sur la classe minoritaire s'accompagne d'une légère augmentation des faux positifs pour la classe majoritaire (20 contre 10 précédemment). Cela peut indiquer un compromis entre les performances sur les deux classes.

En termes globaux, l'AUC reste élevée (0.96), indiquant une capacité continue du modèle à séparer les classes. Cependant, l'impact du SMOTE sur les métriques comme la F1-score, qui équilibre précision et rappel, pourrait être plus révélateur des performances réelles après rééchantillonnage.

Application d'ADASYN (Données Synthétiques)

```
1 ad = ADASYN(random_state=42)
2 X_train_ad, y_train_ad = ad.fit_resample(X_train_scaled, y_train)
3
4 # Distribution après ADASYN
5 plt.figure(figsize=(6,4))
6 sns.countplot(x=y_train_ad)
7 plt.title("Distribution des classes après ADASYN")
8 plt.show()
9
10 model_ad = RandomForestClassifier(random_state=42)
11 model_ad.fit(X_train_ad, y_train_ad)
12 y_pred_ad = model_ad.predict(X_test_scaled)
13
14 print("Performances après ADASYN:")
15 print(classification_report(y_test, y_pred_ad))
16
```

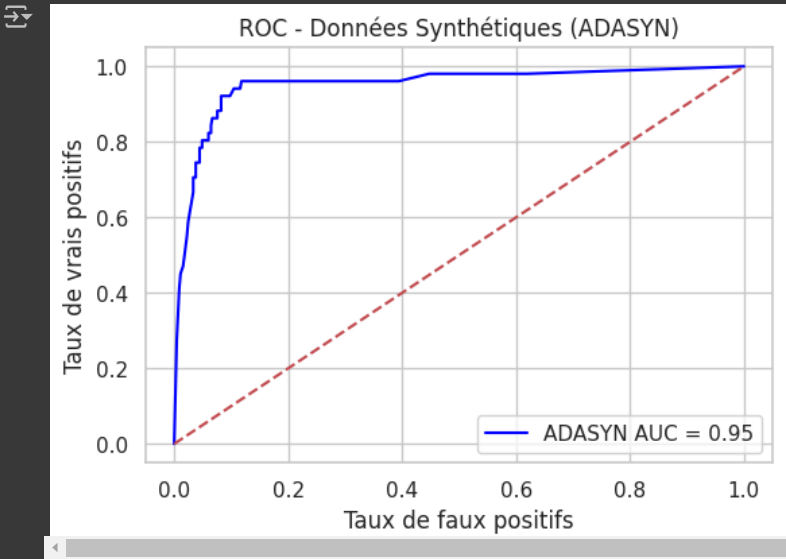


Performances après ADASYN:

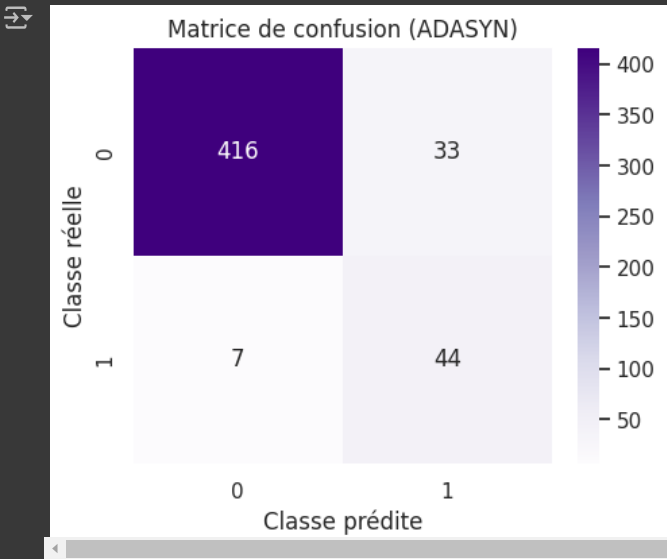
|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.98      | 0.93   | 0.95     | 449     |
| 1            | 0.57      | 0.86   | 0.69     | 51      |
| accuracy     |           |        | 0.92     | 500     |
| macro avg    | 0.78      | 0.89   | 0.82     | 500     |
| weighted avg | 0.94      | 0.92   | 0.93     | 500     |

```
1 fpr_ad, tpr_ad, _ = roc_curve(y_test, model_ad.predict_proba(X_test_scaled)[: ,1])
2 auc_ad = auc(fpr_ad, tpr_ad)
3
4 plt.figure(figsize=(6,4))
5 plt.plot(fpr_ad, tpr_ad, label='ADASYN AUC = %.2f' % auc_ad, color='blue')
6 plt.plot([0,1],[0,1], 'r--')
7 plt.title("ROC - Données Synthétiques (ADASYN)")
8 plt.xlabel("Taux de faux positifs")
9 plt.ylabel("Taux de vrais positifs")
10 plt.legend()
```

```
11 plt.show()
12
```



```
1 cm_ad = confusion_matrix(y_test, y_pred_ad)
2 plt.figure(figsize=(5,4))
3 sns.heatmap(cm_ad, annot=True, fmt='d', cmap='Purples')
4 plt.title("Matrice de confusion (ADASYN)")
5 plt.ylabel("Classe réelle")
6 plt.xlabel("Classe prédite")
7 plt.show()
8
```



Interprétation :

Comme SMOTE, ADASYN vise à améliorer la détection de la classe minoritaire en générant des exemples synthétiques. Cependant, ADASYN se distingue en donnant plus de poids aux zones où les données de la classe minoritaire sont plus difficiles à séparer (zones à faible densité).

Dans ce cas, l'AUC de 0.95 indique une performance légèrement inférieure à celle observée avec SMOTE. Cela peut refléter le fait qu'ADASYN, en se concentrant sur des zones spécifiques, peut introduire davantage de faux positifs pour la classe majoritaire.

L'impact d'ADASYN sur le rappel de la classe minoritaire et les faux positifs de la classe majoritaire nécessite une analyse détaillée à partir de la matrice de confusion et des autres métriques comme la F1-score.

## Application sur le Dataset Titanic

Objectif : Vérifier si les améliorations observées sur les données synthétiques se reproduisent sur un dataset réel déséquilibré.

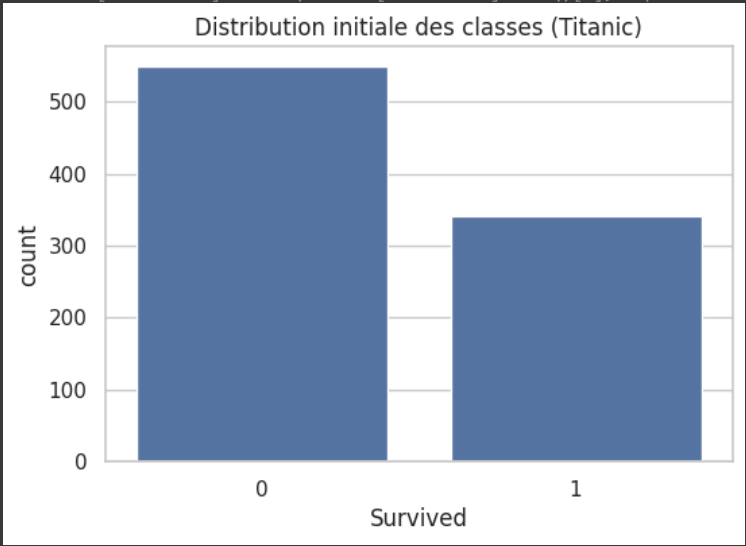
```
1 url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
2 titanic = pd.read_csv(url)
3
4
5 titanic = titanic.drop(['Name', 'Ticket', 'Cabin'], axis=1)
6 titanic['Age'].fillna(titanic['Age'].median(), inplace=True)
7 titanic['Embarked'].fillna(titanic['Embarked'].mode()[0], inplace=True)
8
9 titanic['Sex'] = titanic['Sex'].map({'male':0, 'female':1})
10 titanic = pd.get_dummies(titanic, columns=['Embarked'], drop_first=True)
11
12 X_t = titanic.drop('Survived', axis=1)
13 y_t = titanic['Survived']
14
15 plt.figure(figsize=(6,4))
16 sns.countplot(x=y_t)
17 plt.title("Distribution initiale des classes (Titanic)")
18 plt.show()
19
20 X_train_t, X_test_t, y_train_t, y_test_t = train_test_split(X_t, y_t, stratify=y_t, random_state=42)
21
22 scaler_t = StandardScaler()
23 X_train_t_scaled = scaler_t.fit_transform(X_train_t)
```

```
24 X_test_t_scaled = scaler_t.transform(X_test_t)
25
```

```
<ipython-input-4-88b581e2e973>:6: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment...
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values is a copy.
To avoid this warning in pandas, please use method chaining or `loc` for chained assignments.
For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value, inplace=True)

titanic['Age'].fillna(titanic['Age'].median(), inplace=True)
<ipython-input-4-88b581e2e973>:7: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment...
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values is a copy.
To avoid this warning in pandas, please use method chaining or `loc` for chained assignments.
For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value, inplace=True)

titanic['Embarked'].fillna(titanic['Embarked'].mode()[0], inplace=True)
```



```
1 model_t = RandomForestClassifier(random_state=42)
2 model_t.fit(X_train_t_scaled, y_train_t)
3 y_pred_t = model_t.predict(X_test_t_scaled)
4
5 print("Performances Titanic sans rééchantillonnage:")
6 print(classification_report(y_test_t, y_pred_t))
7
```

|  |           |        |          |         |  |
|--|-----------|--------|----------|---------|--|
| Performances Titanic sans rééchantillonnage: |           |        |          |         |  |
|  | precision | recall | f1-score | support |  |
| 0  | 0.80      | 0.89   | 0.84     | 137     |  |
| 1  | 0.79      | 0.64   | 0.71     | 86      |  |
| accuracy                                     |           |        | 0.79     | 223     |  |
| macro avg                                    | 0.79      | 0.77   | 0.77     | 223     |  |
| weighted avg                                 | 0.79      | 0.79   | 0.79     | 223     |  |

```
1 X_train_t_sm, y_train_t_sm = SMOTE(random_state=42).fit_resample(X_train_t_scaled, y_train_t)
2
3 model_t_sm = RandomForestClassifier(random_state=42)
4 model_t_sm.fit(X_train_t_sm, y_train_t_sm)
5 y_pred_t_sm = model_t_sm.predict(X_test_t_scaled)
6
7 print("Performances Titanic après SMOTE:")
8 print(classification_report(y_test_t, y_pred_t_sm))
9
```

|                                   |           |        |          |         |  |
|-----------------------------------|-----------|--------|----------|---------|--|
| Performances Titanic après SMOTE: |           |        |          |         |  |
|                                   | precision | recall | f1-score | support |  |
| 0                                 | 0.82      | 0.86   | 0.84     | 137     |  |
| 1                                 | 0.76      | 0.70   | 0.73     | 86      |  |
| accuracy                          |           |        | 0.80     | 223     |  |
| macro avg                         | 0.79      | 0.78   | 0.78     | 223     |  |
| weighted avg                      | 0.80      | 0.80   | 0.80     | 223     |  |

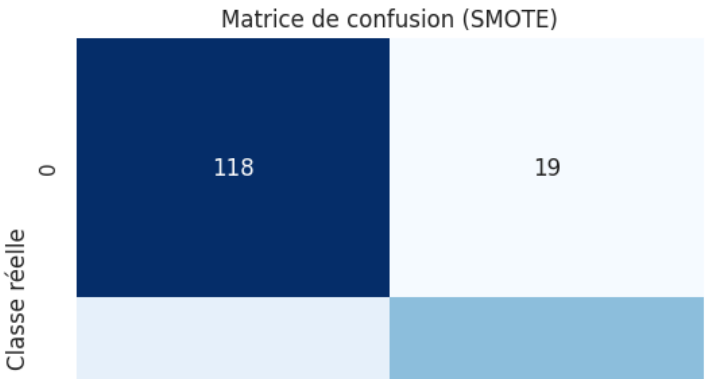
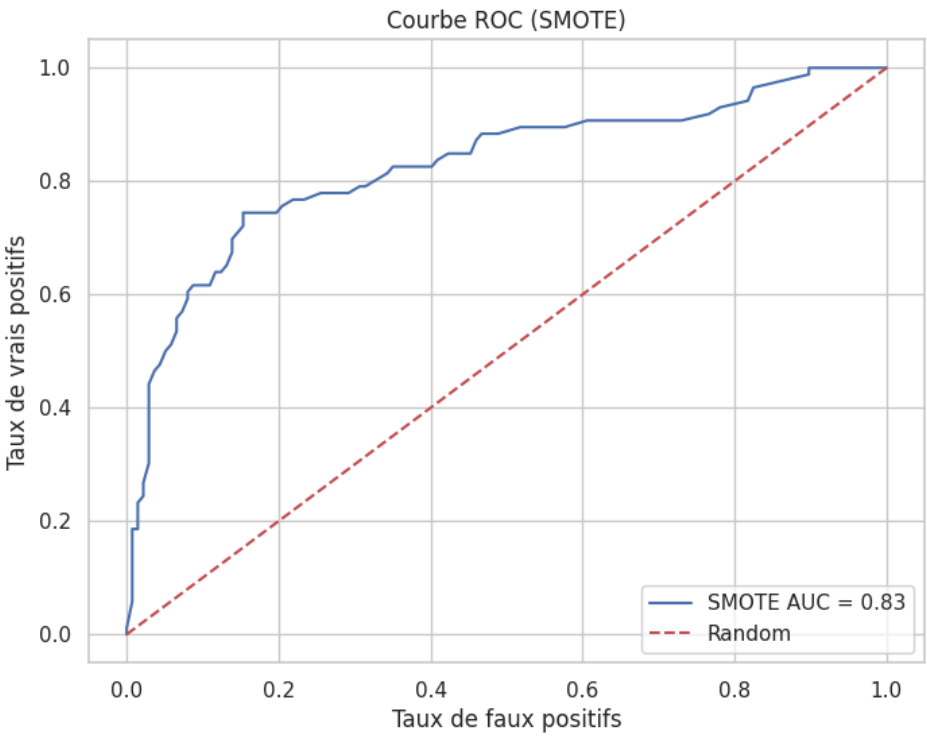
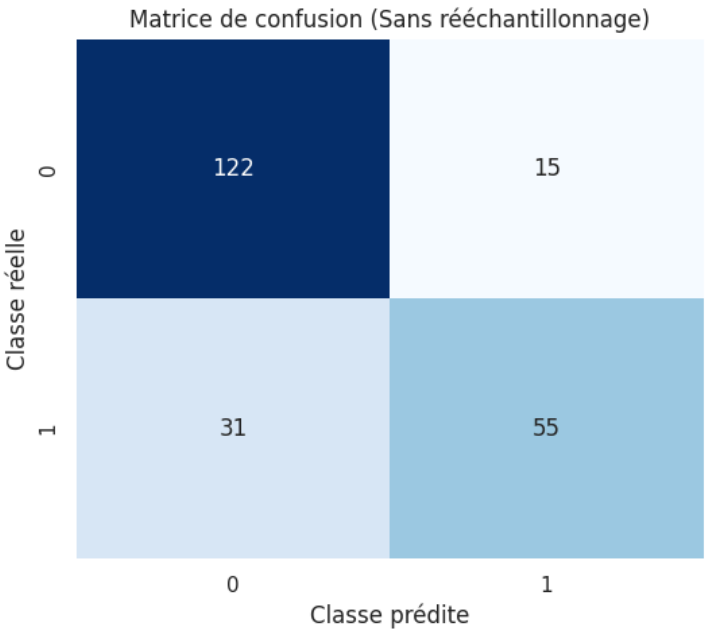
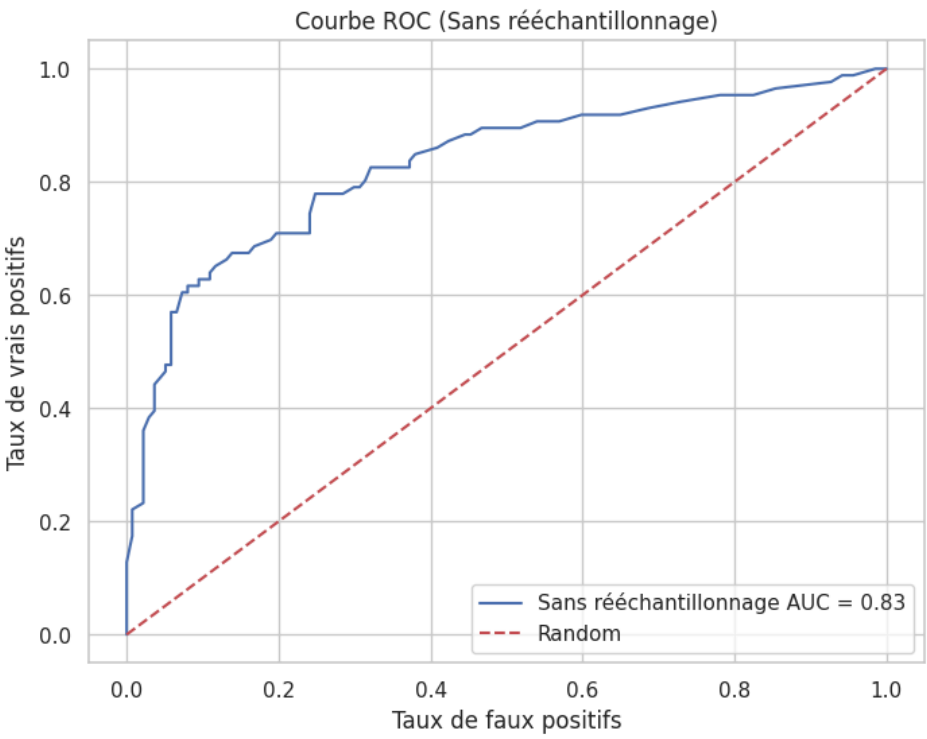
```
1 X_train_t_ad, y_train_t_ad = ADASYN(random_state=42).fit_resample(X_train_t_scaled, y_train_t)
2
3 model_t_ad = RandomForestClassifier(random_state=42)
4 model_t_ad.fit(X_train_t_ad, y_train_t_ad)
5 y_pred_t_ad = model_t_ad.predict(X_test_t_scaled)
6
7 print("Performances Titanic après ADASYN:")
8 print(classification_report(y_test_t, y_pred_t_ad))
9
```

|                                    |           |        |          |         |  |
|------------------------------------|-----------|--------|----------|---------|--|
| Performances Titanic après ADASYN: |           |        |          |         |  |
|                                    | precision | recall | f1-score | support |  |
| 0                                  | 0.81      | 0.82   | 0.81     | 137     |  |
| 1                                  | 0.70      | 0.69   | 0.69     | 86      |  |
| accuracy                           |           |        | 0.77     | 223     |  |
| macro avg                          | 0.75      | 0.75   | 0.75     | 223     |  |
| weighted avg                       | 0.77      | 0.77   | 0.77     | 223     |  |

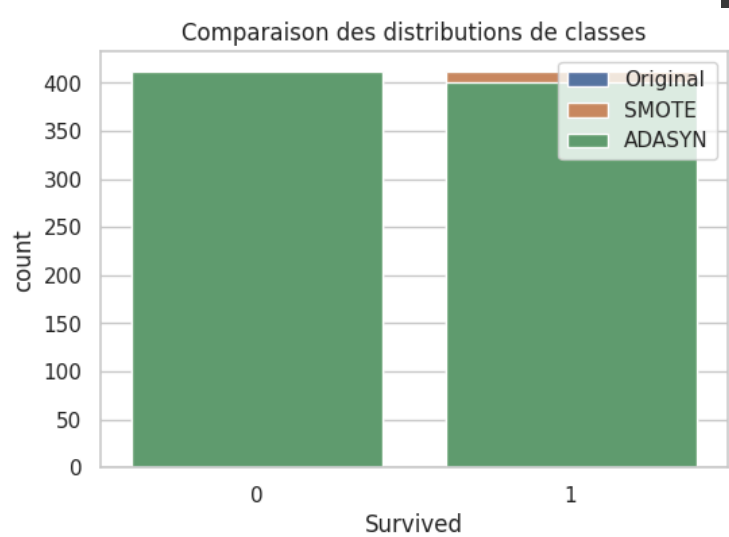
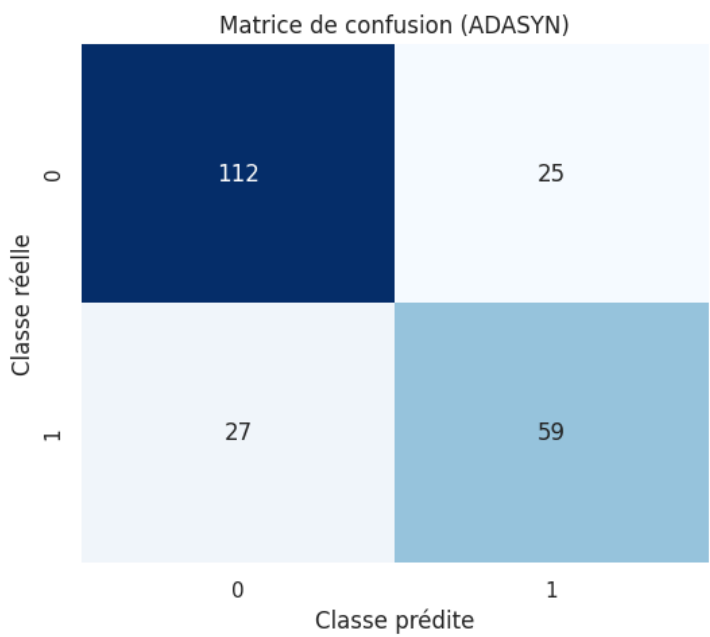
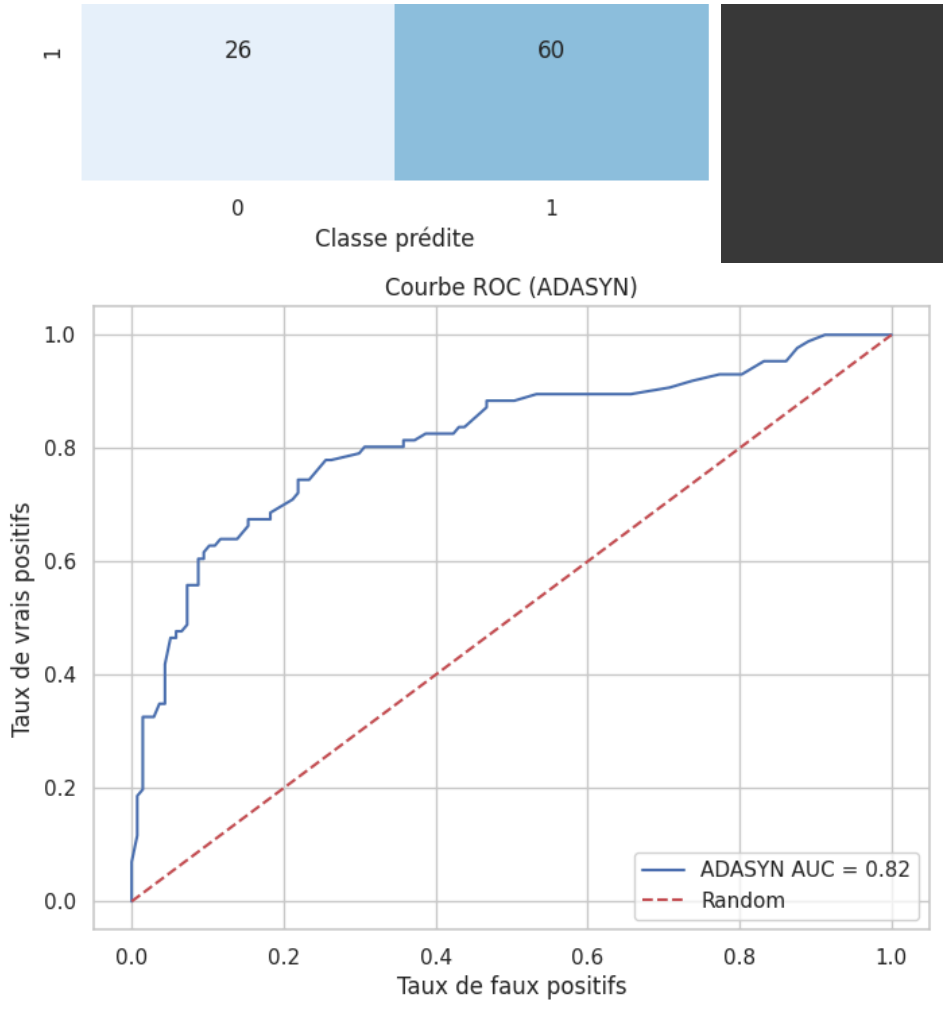
```

1 from sklearn.metrics import roc_curve, auc, confusion_matrix
2 import seaborn as sns
3 import matplotlib.pyplot as plt
4
5 # Fonction pour tracer la courbe ROC
6 def plot_roc_curve(model, X_test, y_test, label):
7     y_pred_proba = model.predict_proba(X_test)[: , 1]
8     fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
9     roc_auc = auc(fpr, tpr)
10
11     plt.plot(fpr, tpr, label=f"{label} AUC = {roc_auc:.2f}")
12     plt.plot([0, 1], [0, 1], 'r--', label="Random")
13     plt.xlabel("Taux de faux positifs")
14     plt.ylabel("Taux de vrais positifs")
15     plt.title(f"Courbe ROC ({label})")
16     plt.legend(loc="lower right")
17
18 # Fonction pour tracer la matrice de confusion
19 def plot_confusion_matrix(y_true, y_pred, title):
20     cm = confusion_matrix(y_true, y_pred)
21     plt.figure(figsize=(6, 5))
22     sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
23     plt.title(title)
24     plt.xlabel("Classe prédite")
25     plt.ylabel("Classe réelle")
26
27 # Visualisation des performances sans rééchantillonnage
28 plt.figure(figsize=(8, 6))
29 plot_roc_curve(model_t, X_test_t_scaled, y_test_t, label="Sans rééchantillonnage")
30 plt.show()
31 plot_confusion_matrix(y_test_t, y_pred_t, title="Matrice de confusion (Sans rééchantillonnage)")
32
33 # Visualisation des performances avec SMOTE
34 plt.figure(figsize=(8, 6))
35 plot_roc_curve(model_t_sm, X_test_t_scaled, y_test_t, label="SMOTE")
36 plt.show()
37 plot_confusion_matrix(y_test_t, y_pred_t_sm, title="Matrice de confusion (SMOTE)")
38
39 # Visualisation des performances avec ADASYN
40 plt.figure(figsize=(8, 6))
41 plot_roc_curve(model_t_ad, X_test_t_scaled, y_test_t, label="ADASYN")
42 plt.show()
43 plot_confusion_matrix(y_test_t, y_pred_t_ad, title="Matrice de confusion (ADASYN)")
44
45 # Comparaison des distributions avant et après rééchantillonnage
46 plt.figure(figsize=(6, 4))
47 sns.countplot(x=y_train_t, label="Original")
48 sns.countplot(x=y_train_t_sm, label="SMOTE")
49 sns.countplot(x=y_train_t_ad, label="ADASYN")
50 plt.title("Comparaison des distributions de classes")
51 plt.legend(["Original", "SMOTE", "ADASYN"])
52 plt.show()
53

```







Interprétation :

1. Performances sans rééchantillonnage :

- Le modèle présente une **AUC de 0.83**, indiquant une performance correcte mais perfectible.
- La matrice de confusion montre une détection modérée des survivants (classe minoritaire) : **55 vrais positifs** contre **31 faux négatifs**.
- Cette performance reflète l'influence du déséquilibre des classes, avec une légère préférence pour la classe majoritaire (non survivants).

2. Performances après SMOTE :

- Avec SMOTE, l'**AUC reste stable à 0.83**, suggérant que le rééchantillonnage n’a pas significativement amélioré la capacité globale du modèle à distinguer les classes.
- La matrice de confusion montre une amélioration notable des prédictions des survivants (classe 1), avec **60 vrais positifs** contre **26 faux négatifs**, au prix d'une légère augmentation des faux positifs (**19 faux positifs** contre **15 précédemment**).

3. Performances après ADASYN :

- L'**AUC diminue légèrement à 0.82**, ce qui pourrait indiquer une légère dégradation de la performance globale.
- La matrice de confusion montre un compromis similaire à SMOTE : **59 vrais positifs** contre **27 faux négatifs**, mais une augmentation des faux positifs (**25**). ADASYN, en se concentrant davantage sur les zones difficiles, semble avoir introduit plus d'erreurs sur la classe majoritaire.

4. Comparaison des distributions :

- Les graphiques de distribution confirment que SMOTE et ADASYN ont équilibré les classes dans l’ensemble d’entraînement, ce qui améliore le rappel pour la classe minoritaire (survivants) au détriment d’une légère baisse de précision.

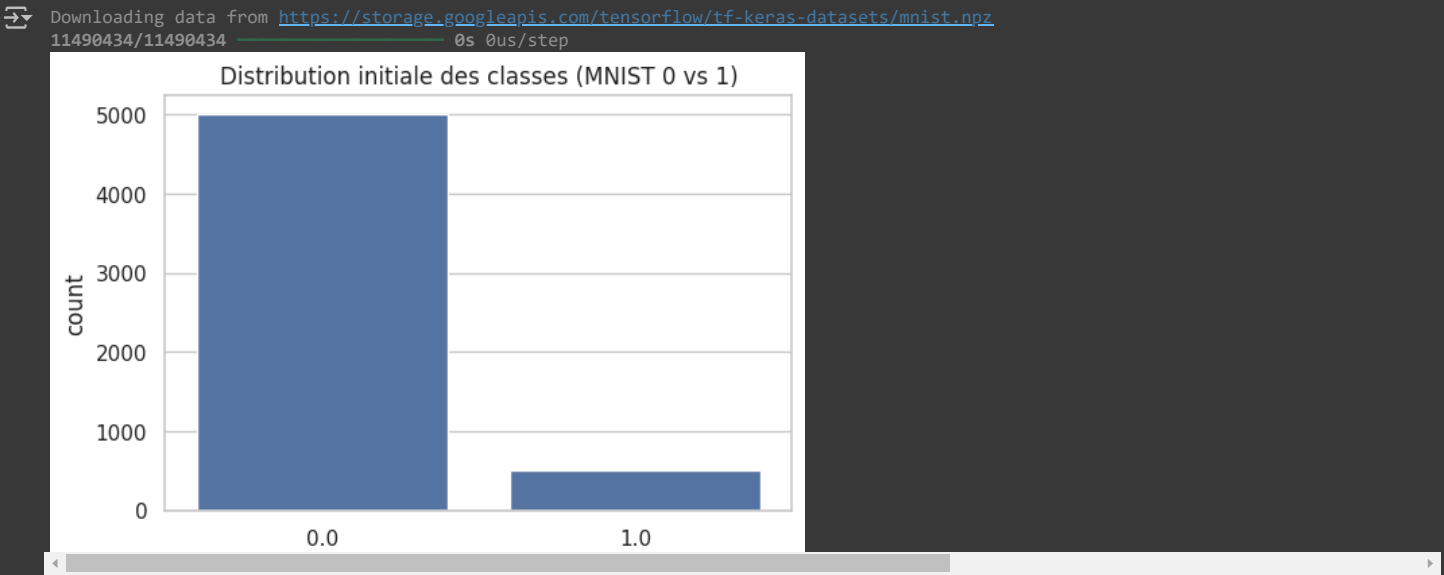
Conclusion générale :

- Le Titanic étant **moins déséquilibré** et plus complexe que le dataset synthétique, les améliorations apportées par SMOTE et ADASYN sont **modestes**.
- SMOTE et ADASYN ont amélioré le rappel pour la classe minoritaire (survivants) au prix d'une légère augmentation des faux positifs, ce qui est attendu avec de telles techniques.
- Ces résultats soulignent l’importance de choisir un rééchantillonnage adapté en fonction de la nature du dataset. Une optimisation supplémentaire des hyperparamètres du modèle ou une exploration d'autres techniques de gestion des données déséquilibrées (ex. : pondération des classes) pourrait améliorer les performances.

✓ Application sur MNIST (Classes 0 et 1 Déséquilibrées)

Objectif : Simuler un déséquilibre en ne gardant que les classes '0' et '1' de MNIST, avec beaucoup de '0' et très peu de '1', puis appliquer SMOTE/ADASYN.

```
1 (X_mnist_train, y_mnist_train), (X_mnist_test, y_mnist_test) = mnist.load_data()
2
3 # Filtrage pour n'avoir que les classes 0 et 1
4 train_filter = (y_mnist_train == 0) | (y_mnist_train == 1)
5 test_filter = (y_mnist_test == 0) | (y_mnist_test == 1)
6
7 X_mnist_train, y_mnist_train = X_mnist_train[train_filter], y_mnist_train[train_filter]
8 X_mnist_test, y_mnist_test = X_mnist_test[test_filter], y_mnist_test[test_filter]
9
10 # Réduction du nombre d'exemples de '1'
11 X_mnist_train_0 = X_mnist_train[y_mnist_train==0][:5000]
12 X_mnist_train_1 = X_mnist_train[y_mnist_train==1][:500]
13 y_mnist_train_0 = np.zeros(len(X_mnist_train_0))
14 y_mnist_train_1 = np.ones(len(X_mnist_train_1))
15
16 X_mnist_train = np.concatenate((X_mnist_train_0, X_mnist_train_1), axis=0)
17 y_mnist_train = np.concatenate((y_mnist_train_0, y_mnist_train_1), axis=0)
18
19 # Mise à plat des images
20 X_mnist_train = X_mnist_train.reshape((X_mnist_train.shape[0], -1))/255.0
21 X_mnist_test = X_mnist_test.reshape((X_mnist_test.shape[0], -1))/255.0
22
23 plt.figure(figsize=(6,4))
24 sns.countplot(x=y_mnist_train)
25 plt.title("Distribution initiale des classes (MNIST 0 vs 1)")
26 plt.show()
27
```



```
1 X_train_m, X_val_m, y_train_m, y_val_m = train_test_split(X_mnist_train, y_mnist_train, stratify=y_mnist_train, random_state=42)
2
3 model_m = RandomForestClassifier(random_state=42)
4 model_m.fit(X_train_m, y_train_m)
5 y_pred_m = model_m.predict(X_val_m)
6
7 print("Performances MNIST sans rééchantillonnage:")
8 print(classification_report(y_val_m, y_pred_m))
9
```

↗

| Performances MNIST sans rééchantillonnage: |           |        |          |         |      |
|--|-----------|--------|----------|---------|------|
|  | precision | recall | f1-score | support |      |
|  | 0.0       | 1.00   | 1.00     | 1.00    | 1250 |
|  | 1.0       | 0.99   | 0.99     | 0.99    | 125  |
|  |           |        |          |         |      |
| accuracy                                   |           |        |          | 1.00    | 1375 |
| macro avg                                  | 1.00      | 1.00   | 1.00     | 1.00    | 1375 |
| weighted avg                               | 1.00      | 1.00   | 1.00     | 1.00    | 1375 |

```
1 X_train_m_sm, y_train_m_sm = SMOTE(random_state=42).fit_resample(X_train_m, y_train_m)
2
3 model_m_sm = RandomForestClassifier(random_state=42)
4 model_m_sm.fit(X_train_m_sm, y_train_m_sm)
5 y_pred_m_sm = model_m_sm.predict(X_val_m)
6
7 print("Performances MNIST après SMOTE:")
8 print(classification_report(y_val_m, y_pred_m_sm))
9
```

↗

| Performances MNIST après SMOTE: |           |        |          |         |      |
|---------------------------------|-----------|--------|----------|---------|------|
|                                 | precision | recall | f1-score | support |      |
|                                 | 0.0       | 1.00   | 1.00     | 1.00    | 1250 |
|                                 | 1.0       | 1.00   | 0.99     | 1.00    | 125  |
|                                 |           |        |          |         |      |
| accuracy                        |           |        |          | 1.00    | 1375 |
| macro avg                       | 1.00      | 1.00   | 1.00     | 1.00    | 1375 |
| weighted avg                    | 1.00      | 1.00   | 1.00     | 1.00    | 1375 |

```
1 X_train_m_ad, y_train_m_ad = ADASYN(random_state=42).fit_resample(X_train_m, y_train_m)
2
3 model_m_ad = RandomForestClassifier(random_state=42)
4 model_m_ad.fit(X_train_m_ad, y_train_m_ad)
5 y_pred_m_ad = model_m_ad.predict(X_val_m)
6
7 print("Performances MNIST après ADASYN:")
8 print(classification_report(y_val_m, y_pred_m_ad))
9
```

↗

| Performances MNIST après ADASYN: |           |        |          |         |      |
|----------------------------------|-----------|--------|----------|---------|------|
|                                  | precision | recall | f1-score | support |      |
|                                  | 0.0       | 1.00   | 1.00     | 1.00    | 1250 |
|                                  | 1.0       | 1.00   | 0.99     | 1.00    | 125  |
|                                  |           |        |          |         |      |
| accuracy                         |           |        |          | 1.00    | 1375 |
| macro avg                        | 1.00      | 1.00   | 1.00     | 1.00    | 1375 |
| weighted avg                     | 1.00      | 1.00   | 1.00     | 1.00    | 1375 |

Interprétation :

1. Performances sans rééchantillonnage :

- Le modèle atteint une **précision, un rappel et un F1-score presque parfaits (1.00)** pour la classe majoritaire (0) et la classe minoritaire (1).
- Malgré le déséquilibre important entre les classes (10 fois plus de 0 que de 1), le modèle parvient à bien détecter les 1 grâce à la simplicité des données (MNIST 0 et 1 sont bien séparables).
- La précision globale est de 1.00, ce qui reflète une performance exceptionnelle sans rééchantillonnage.

2. Performances après SMOTE :

- Les métriques restent globalement **identiques à celles obtenues sans rééchantillonnage**, avec une précision, un rappel et un F1-score quasi parfaits.
- Le rappel pour la classe 1 (0.99) montre une légère stabilité, mais l’impact de SMOTE est minime dans ce cas car le modèle performait déjà très bien sans rééchantillonnage.

3. Performances après ADASYN :

- Les résultats après ADASYN sont également identiques à ceux obtenus avec SMOTE, avec une **précision globale de 1.00** et des métriques presque parfaites pour les deux classes.
- Tout comme avec SMOTE, ADASYN n’apporte pas de gain significatif dans ce contexte, probablement en raison de la simplicité des données et de la bonne séparation initiale des classes 0 et 1.

Conclusion :


- **L’impact de SMOTE et ADASYN est négligeable** dans ce scénario car le modèle performe déjà extrêmement bien sans rééchantillonnage. Cela est dû à :
  - La **clarté des données MNIST** pour les classes 0 et 1, qui sont naturellement bien séparables.
  - Le **déséquilibre limité** (10:1) qui n’est pas assez sévère pour que le modèle nécessite un rééchantillonnage pour apprendre correctement.
- Dans des cas où les données sont plus complexes ou où le déséquilibre est plus extrême, SMOTE et ADASYN pourraient être plus bénéfiques. Ici, leur effet est superflu.

✓ Ajustements des Paramètres (Grid Search sur SMOTE/ADASYN & RandomForest)

Nous allons maintenant démontrer l’impact de l’ajustement des paramètres `k_neighbors` (SMOTE/ADASYN) et `n_estimators` (RandomForest) sur le dataset synthétique.

```
1 # On reprend le dataset synthétique initial
2 X, y = make_classification(n_samples=2000,
3                             n_features=2,
4                             n_informative=2,
5                             n_redundant=0,
6                             n_clusters_per_class=1,
7                             weights=[0.9, 0.1],
8                             class_sep=1.0,
9                             random_state=42)
10
11 X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
12 scaler = StandardScaler()
13 X_train_scaled = scaler.fit_transform(X_train)
14 X_test_scaled = scaler.transform(X_test)
15
```

```
1 pipe_sm = Pipeline([
2     ('smote', SMOTE()),
3     ('clf', RandomForestClassifier(random_state=42))
4 ])
5
6 param_grid_sm = {
7     'smote_k_neighbors': [2, 5, 10],
8     'clf_n_estimators': [50, 100, 200]
9 }
10
11 grid_sm = GridSearchCV(pipe_sm, param_grid_sm, scoring='f1_macro', cv=3, n_jobs=-1)
12 grid_sm.fit(X_train_scaled, y_train)
13
14 print("Meilleurs paramètres SMOTE:", grid_sm.best_params_)
15 print("Meilleur score (F1 macro) SMOTE:", grid_sm.best_score_)
16
17 y_pred_grid_sm = grid_sm.predict(X_test_scaled)
18 print("Rapport de classification SMOTE (paramètres optimisés):")
19 print(classification_report(y_test, y_pred_grid_sm))
20
```



Meilleurs paramètres SMOTE: {'clf\_n\_estimators': 200, 'smote\_k\_neighbors': 10}


Meilleur score (F1 macro) SMOTE: 0.8618045114932248

Rapport de classification SMOTE (paramètres optimisés):

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.99      | 0.95   | 0.97     | 449     |
| 1            | 0.69      | 0.90   | 0.78     | 51      |
| accuracy     |           |        | 0.95     | 500     |
| macro avg    | 0.84      | 0.93   | 0.88     | 500     |
| weighted avg | 0.96      | 0.95   | 0.95     | 500     |

```
1 pipe_ad = Pipeline([
2     ('adasyn', ADASYN()),
3     ('clf', RandomForestClassifier(random_state=42))
4 ])
5
6 param_grid_ad = {
7     'adasyn_n_neighbors': [2, 5, 10],
8     'clf_n_estimators': [50, 100, 200]
9 }
```

```
10
11 grid_ad = GridSearchCV(pipe_ad, param_grid_ad, scoring='f1_macro', cv=3, n_jobs=-1)
12 grid_ad.fit(X_train_scaled, y_train)
13
14 print("Meilleurs paramètres ADASYN:", grid_ad.best_params_)
15 print("Meilleur score (F1 macro) ADASYN:", grid_ad.best_score_)
16
17 y_pred_grid_ad = grid_ad.predict(X_test_scaled)
18 print("Rapport de classification ADASYN (paramètres optimisés):")
19 print(classification_report(y_test, y_pred_grid_ad))
20
```

Meilleurs paramètres ADASYN: {'adasyn\_\_n\_neighbors': 10, 'clf\_\_n\_estimators': 100}  
Meilleur score (F1 macro) ADASYN: 0.8405781174332212  
Rapport de classification ADASYN (paramètres optimisés):

|              | precision | recall | f1-score | support |     |
|--------------|-----------|--------|----------|---------|-----|
|              | 0         | 0.99   | 0.93     | 0.96    | 449 |
|              | 1         | 0.59   | 0.92     | 0.72    | 51  |
|              |           |        |          |         |     |
| accuracy     |           |        |          | 0.93    | 500 |
| macro avg    | 0.79      | 0.93   | 0.84     |         | 500 |
| weighted avg | 0.95      | 0.93   | 0.93     |         | 500 |

Interprétation :

## 1. Résultats optimisés avec SMOTE :

- Meilleurs paramètres :
  - clf\_\_n\_estimators = 200
  - smote\_\_k\_neighbors = 10
- Meilleur score F1 (macro) : 0.8618
- Rapport de classification :
  - Classe majoritaire (0) :
    - Précision : 0.99
    - Rappel : 0.95
    - F1-score : 0.97
  - Classe minoritaire (1) :
    - Précision : 0.69
    - Rappel : 0.90
    - F1-score : 0.78
  - Précision globale (accuracy) : 0.95
  - La classe minoritaire bénéficie d'un **rappel élevé (0.90)**, ce qui montre l'efficacité de SMOTE pour équilibrer les classes et améliorer la détection des échantillons de la classe minoritaire.

## 2. Résultats optimisés avec ADASYN :

- Meilleurs paramètres :
  - clf\_\_n\_estimators = 100
  - adasyn\_\_n\_neighbors = 10
- Meilleur score F1 (macro) : 0.8406
- Rapport de classification :
  - Classe majoritaire (0) :
    - Précision : 0.99
    - Rappel : 0.93
    - F1-score : 0.96
  - Classe minoritaire (1) :
    - Précision : 0.59
    - Rappel : 0.92
    - F1-score : 0.72
  - Précision globale (accuracy) : 0.93
  - ADASYN montre également une **amélioration significative du rappel pour la classe minoritaire (0.92)**, bien que sa précision soit plus faible (0.59), indiquant une légère augmentation des faux positifs.

## 3. Conclusion :

- Le choix des hyperparamètres (k\_neighbors pour SMOTE/ADASYN et n\_estimators pour RandomForest) influence directement les performances :
  - SMOTE a produit un meilleur compromis avec une précision et un F1-score légèrement supérieurs pour la classe minoritaire, tout en maintenant une précision globale élevée.
  - ADASYN a légèrement amélioré le rappel pour la classe minoritaire, mais au prix d'une précision réduite, introduisant plus de faux positifs.
- Impact des paramètres :
  - k\_neighbors dans SMOTE/ADASYN détermine la proximité des échantillons synthétiques à la classe minoritaire réelle, influençant ainsi la qualité des données générées.

- **n\_estimators** dans RandomForest affecte la stabilité et la performance globale du modèle, avec des valeurs plus élevées favorisant souvent de meilleures performances.

## Interprétation générale

### Observations Générales :

- **Données synthétiques :**
  - L'application de SMOTE et d'ADASYN permet d'augmenter sensiblement le **rappel de la classe minoritaire** et d'améliorer les scores (F1, AUC).
  - Les heatmaps montrent une **meilleure répartition des prédictions** sur les deux classes.
- **Titanic :**
  - L'amélioration est présente mais **moins spectaculaire**.
  - Le dataset réel, plus complexe, ne se laisse pas toujours facilement rééquilibrer.