

REPUBLIQUE DU SENEGAL



UNIVERSITE CHEIKH ANTA DIOP DE DAKAR

Département de Mathématiques-Informatique

Section Informatique



PROJET PERFORMANCE: SIMULATION ET PRÉDICTION DU TEMPS D'ATTENTE (VANAD)

Présenté par :

Ouly TOURE

Professeur:

Dr Mamadou Thiongane

Année: 2024-2025

Table des matières

Introduction	1
Chargement et regroupement des fichiers CSV:	2
Nombre d'appels par heure:	7
Nombre d'appels par jour de la semaine:	7
Nombre d'appels par type (file d'attente):	8
Top 10 des services les plus demandés:	8
Top 10 des services les moins demandés:	9
Top 10 des agents les plus actifs:	9
Top 10 des agents les moins actifs:	10
Distribution du nombre d'appels par agent:	10
Évolution du nombre d'appels par jour:	11
Volume d'appels par heure et jour de la semaine (heatmap):	11
Évolution mensuelle du volume d'appels:	12
Top 10 des journées les plus chargées:	12
Durée moyenne d'activité par agent (Top 10):	13
Temps moyen d'attente par heure:	13
Pourcentage d'appels pris en charge rapidement:	14
Variabilité des temps d'attente par file d'attente:	14
Taux de réponses rapides (< 1 minute) par heure	14
Répartition du volume d'appels par agent et file d'attente (Top 10 agents):	15
Distribution des temps d'attente des appels:	16
II. Conception et implémentation Java (simulation)	16
2. Explication des classes:	18
Rôles et fonctions principales :	18
Utilisation dans le projet VANAD :	19
Rôles principaux :	26
Fonctionnalités principales :	26
Méthodes utilitaires internes :	27
Utilisation dans le projet :	27

Objectifs principaux de MoteurReplay :	30
Structure interne :	31
Principales méthodes :	31
Autres méthodes :	32
Objectifs de la classe :	33
Attributs principaux :	33
Méthodes principales :	34
chargerEtPreparerDonnees(...)	34
initialiserMoteurReplay()	35
executerReplayEvenementParEvenement()	35
exporterDonneesEntrainement()	35
analyserResultatsSimulation()	35
validerPredicteurs()	36
But de la classe :	37
Rôles principaux :	38
Fonctionnalités principales :	38
Attributs clés :	39
Utilisation dans le projet :	39
Amélioration de la lisibilité des données avec Excel	43
IV. Analyse exploratoire des données générées	44
V. Modélisation par réseau de neurones (ANN)	46
Importation des bibliothèques nécessaires	46
Chargement du fichier de données généré par la simulation Java	47
Préparation des données pour l'entraînement	48
80 % pour l'entraînement (X_train, y_train).....	48
20 % pour le test (X_test, y_test)	48
Création du modèle de réseau de neurones (ANN)	49
Entraînement du modèle	50
VI. Comparaison des modèles	54
Conclusion	55

Introduction

Le présent rapport décrit le projet de simulation et de prédition que nous avons réalisé, visant à évaluer l'efficacité de différents prédicteurs de délai d'attente dans un centre d'appel multicompétences à partir de données réelles. Le travail s'est articulé en deux grandes parties : un traitement exploratoire et statistique des données dans un environnement Python (Notebook), puis le développement d'une application Java pour simuler le fonctionnement du centre et générer un jeu de données structuré pour l'apprentissage automatique.

Les données utilisées proviennent du centre d'appel VANAD basé à Rotterdam (Pays-Bas) et couvrent l'année 2014. Elles incluent un volume important d'appels clients associés aux informations de service et de traitement. L'objectif était de reconstituer dynamiquement l'état du système à chaque arrivée d'appel, afin de pouvoir tester trois types de prédicteurs du délai d'attente : LES, Avg-LES et un réseau de neurones artificiel (ANN).

Ce projet nous a permis de mettre en œuvre nos compétences en modélisation stochastique, en programmation orientée objet avec Java, en structuration de jeux de données pour l'IA, ainsi qu'en visualisation statistique et en apprentissage supervisé. L'application Java repose sur un moteur de *replay* temporel fidèle, intégrant les appels et les activités pour simuler l'évolution de l'état du système au fil du temps.

Dans les sections suivantes, nous détaillerons d'abord la phase de préparation des données dans le notebook Python, puis l'implémentation complète de la simulation Java et la génération du fichier d'apprentissage. Enfin, nous décrirons les étapes prévues pour entraîner le modèle ANN, comparer les prédicteurs entre eux, et évaluer les performances du système.

I. Préparation et exploration des données (Notebook Python)

Pour réaliser la préparation et l'exploration des données, j'ai utilisé **JupyterLab**. Le travail a été effectué dans un notebook Python interactif, permettant à la fois l'écriture du code, l'exécution pas à pas et la visualisation immédiate des résultats.

➤ **Importation des bibliothèques nécessaires:**

```
[1]: # 1. Importation des bibliothèques nécessaires
import pandas as pd
import numpy as np
import glob
import os
from datetime import datetime, time
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error

sns.set(style="whitegrid")
```



Dans cette première étape, nous importons l'ensemble des bibliothèques Python utilisées pour le traitement et l'analyse des données :

- ◆ **pandas, numpy** : manipulation et transformation de données tabulaires.
- ◆ **glob, os** : gestion automatique des fichiers (lecture par lot de tous les fichiers CSV mensuels).
- ◆ **datetime, time** : traitement des horodatages.
- ◆ **matplotlib.pyplot, seaborn** : création de graphiques et visualisations statistiques.
- ◆ **sklearn.model_selection, sklearn.neural_network, sklearn.metrics** : séparation du jeu de données, entraînement du modèle de réseau de neurones (ANN) et évaluation des performances (MSE).

La commande `sns.set(style="whitegrid")` permet d'améliorer le style des graphiques générés avec Seaborn.

➤ **Chargement et regroupement des fichiers CSV:**

```
[2]: # 2. Chargement des fichiers CSV

# chemin des fichiers
chemin_base = r"C:\Users\DELL\Desktop\M2_BI_2025\Performance_des_SD\VANAD_data"

# Recherche des fichiers d'appels et d'activités
fichiers_appels = sorted(glob.glob(f"{chemin_base}\calls-2014-*.csv"))
fichiers_activites = sorted(glob.glob(f"{chemin_base}\activity-2014-*.csv"))

# Affichage de contrôle
print(f"{len(fichiers_appels)} fichiers d'appels trouvés.")
print(f"{len(fichiers_activites)} fichiers d'activités trouvés.")

# Concaténation des appels (de janvier à décembre 2014)
donnees_appels = pd.concat([pd.read_csv(f) for f in fichiers_appels], ignore_index=True)

# Concaténation des activités des agents (de janvier à décembre 2014)
donnees_activites = pd.concat([pd.read_csv(f) for f in fichiers_activites], ignore_index=True)
```

Dans cette étape, tous les fichiers de données du centre d'appel VANAD sont automatiquement chargés et regroupés :

- ◆ **Définition du chemin** où se trouvent les fichiers CSV contenant les journaux d'appels (calls-2014-*.csv) et d'activités d'agents (activity-2014-*.csv) pour chaque mois de l'année 2014.
- ◆ Utilisation de glob.glob(...) pour **lister automatiquement** tous les fichiers mensuels correspondant aux appels et aux activités.
- ◆ Utilisation de pandas.concat(...) pour **fusionner** tous les fichiers d'appels et tous les fichiers d'activités en deux **dataframes complets** : donnees_appels et donnees_activites, couvrant l'ensemble de l'année 2014.

Cela permet de centraliser les données en un seul bloc pour faciliter le traitement et la visualisation.

➤ Nettoyage et transformation des données d'appels:

```
[3]: # 3. Nettoyage des données d'appels
def nettoyage_et_conversion_appels(df):
    # Afficher les colonnes initiales pour vérification
    print("Colonnes avant nettoyage:", df.columns.tolist())

    # Normaliser noms de colonnes : strip, Lowercase, underscore (optionnel si nécessaire)
    df.columns = df.columns.str.strip().str.lower().str.replace(" ", "_")

    # Conversion forcée des colonnes datetime, en coercer pour gérer erreurs
    for col in ['date_received', 'answered', 'hangup']:
        df[col] = pd.to_datetime(df[col], errors='coerce')

    # Supprimer les lignes où une des dates critiques est manquante (NaN)
    df.dropna(subset=['date_received', 'answered', 'hangup'], inplace=True)

    # Calcul des temps d'attente (waiting_time) et de service (service_time) en secondes
    df['waiting_time'] = (df['answered'] - df['date_received']).dt.total_seconds()
    df['service_time'] = (df['hangup'] - df['answered']).dt.total_seconds()

    # Supprimer les lignes où waiting_time ou service_time est négatif (anomalies)
    df = df[(df['waiting_time'] >= 0) & (df['service_time'] >= 0)]

    # Extraction des composantes temporelles depuis date_received
    df['year'] = df['date_received'].dt.year
    df['month'] = df['date_received'].dt.month
    df['day'] = df['date_received'].dt.day
    df['day_of_week'] = df['date_received'].dt.dayofweek
    df['hour'] = df['date_received'].dt.hour
    df['minute'] = df['date_received'].dt.minute
    df['second'] = df['date_received'].dt.second
    df['time_of_day'] = df['hour'] + df['minute']/60 + df['second']/3600

    # Filtrer pour ne garder que les appels en jours ouvrables (lun-ven) et heures 08h-20h
    df = df[(df['day_of_week'] < 5) & (df['hour'] >= 8) & (df['hour'] <= 20)]

    # Tri chronologique
    df = df.sort_values(by='date_received').reset_index(drop=True)

    # Affichage post nettoyage pour contrôle
    print(f"Nombre de lignes après nettoyage : {len(df)}")
    print(f"Colonnes après nettoyage : {df.columns.tolist()}")

    return df

# Application du nettoyage à mes données appels
donnees_appels = nettoyage_et_conversion_appels(donnees_appels)
```

Dans cette étape, les données brutes des appels du centre d'appel VANAD sont nettoyées et préparées pour l'analyse :

- ◆ Les noms des colonnes sont normalisés (espaces supprimés, tout en minuscules, et format underscore) pour garantir une manipulation cohérente et éviter les erreurs liées aux variations de noms.
- ◆ Les colonnes contenant des dates et heures critiques (date_received, answered, hangup) sont converties en format datetime, avec une gestion robuste des erreurs (les valeurs invalides sont transformées en valeurs manquantes).
- ◆ Les lignes où une des dates critiques est absente sont supprimées, afin de ne conserver que les enregistrements complets et fiables.

- ◆ Deux variables sont calculées : le temps d'attente (`waiting_time`) entre la réception et la prise en charge de l'appel, ainsi que le temps de service (`service_time`) entre la prise en charge et la fin de l'appel, tous deux exprimés en secondes.
- ◆ Les enregistrements avec des durées négatives, qui indiqueraient des anomalies ou erreurs, sont également éliminés.
- ◆ Plusieurs composantes temporelles sont extraites de la date de réception pour faciliter les analyses temporelles : année, mois, jour, jour de la semaine, heure, minute, seconde, ainsi qu'une variable continue représentant l'heure fractionnaire dans la journée (`time_of_day`).
- ◆ Un filtrage est appliqué pour ne conserver que les appels reçus les jours ouvrables (du lundi au vendredi) et durant les plages horaires opérationnelles de 8h à 20h.
- ◆ Enfin, les données sont triées chronologiquement selon la date de réception des appels, ce qui facilite les traitements séquentiels et la modélisation.

Cette étape garantit que les données d'appels sont propres, cohérentes et prêtes pour les étapes ultérieures d'analyse, de visualisation et de modélisation prédictive.

Pour le nettoyage des données d'activités, la démarche est similaire à celle appliquée aux appels :

- ◆ Normalisation des noms de colonnes,
- ◆ Conversion des dates de début et fin d'activité en format datetime,
- ◆ Suppression des enregistrements incomplets ou contenant des durées négatives ou nulles,
- ◆ Extraction des variables temporelles (année, mois, jour, jour de la semaine, heure, minute, seconde, heure fractionnaire),
- ◆ Filtrage des données pour ne conserver que les activités réalisées les jours ouvrables entre 8h et 20h,
- ◆ Tri chronologique des données selon la date de début d'activité.

Cette méthode assure une cohérence et une qualité des données avant leur exploitation.

➤ Sauvegarde des données nettoyées:

```
[7]: # Dossier de sauvegarde existant
chemin_projet = r"C:\Users\DELL\Desktop\M2_BI_2025\Performance_des_SD\Projet_Perf_Ouly_TOURE_M2_BI_2025\SimulationCentreApp

# Chemins complets des fichiers à enregistrer
chemin_appels = os.path.join(chemin_projet, "donnees_appels_2014_nettoyees.csv")
chemin_activites = os.path.join(chemin_projet, "donnees_activites_2014_nettoyees.csv")

# Sauvegarde des fichiers CSV nettoyés
donnees_appels.to_csv(chemin_appels, index=False)
donnees_activites.to_csv(chemin_activites, index=False)

# Message de confirmation
print("Sauvegarde réussie :")
print(f" - Appels : {chemin_appels}")
print(f" - Activités : {chemin_activites}")
```

Après nettoyage et préparation, les jeux de données d'appels et d'activités sont sauvegardés au format CSV dans un dossier spécifique du projet.

- ◆ Le chemin de destination est défini de manière centralisée dans la variable `chemin_projet`.
- ◆ Les fichiers nettoyés sont enregistrés sous des noms explicites (`donnees_appels_2014_nettoyees.csv` et `donnees_activites_2014_nettoyees.csv`) pour assurer une bonne traçabilité.
- ◆ La méthode `to_csv()` de pandas est utilisée pour écrire les dataframes dans ces fichiers sans inclure l'index.
- ◆ Un message de confirmation affiche le succès de la sauvegarde et les chemins complets des fichiers générés.

Cette étape garantit la persistance des données préparées pour une utilisation ultérieure dans l'analyse ou la modélisation.

➤ Visualisations:

Pour une meilleure compréhension et une visualisation claire des statistiques, nous vous

invitons à consulter le fichier **notebook_Ouly_TOURE_M2BI_2025.ipynb**, où l'ensemble des graphiques et analyses visuelles sont présentés de manière interactive.

❖ Nombre d'appels par heure:

```
[9]: # Nombre d'appels par heure (08h à 20h)
# Groupement par heure (tous jours confondus)
appels_par_heure = donnees_appels.groupby('hour').size().reset_index(name='nombre_appels')

# Affichage en texte
print("Nombre d'appels par heure :")
print(appels_par_heure)

plt.figure(figsize=(10,6))
sns.barplot(data=appels_par_heure, x='hour', y='nombre_appels', color='skyblue')

plt.title("Nombre d'appels par heure")
plt.xlabel("Heure de la journée")
plt.ylabel("Nombre d'appels")

plt.show()
```

Cette étape consiste à analyser la répartition horaire des appels reçus au centre d'appel VANAD :

- ◆ Les données d'appels nettoyées sont regroupées selon l'heure de réception (hour), en cumulant le nombre total d'appels par heure, toutes journées confondues.
- ◆ Le résultat est un tableau synthétique qui indique combien d'appels ont été reçus pour chaque heure de la journée.
- ◆ Une visualisation graphique est réalisée à l'aide de la bibliothèque seaborn sous forme d'un histogramme à barres.
- ◆ Ce graphique met en évidence la variation du volume d'appels au fil des heures, ce qui peut aider à identifier les plages horaires de forte activité.
- ◆ Les axes sont clairement annotés, et le graphique est titré pour faciliter la compréhension.

Cette représentation permet de mieux comprendre la dynamique journalière du centre d'appel et oriente l'analyse des performances et des ressources nécessaires.

❖ Nombre d'appels par jour de la semaine:

```
[10]: # Nombre d'appels par jour (lundi au vendredi)
# Création de la colonne 'jour_semaine' avec noms complets des jours en anglais
donnees_appels['jour_semaine'] = donnees_appels['date_received'].dt.day_name()

# Ordre des jours de la semaine à afficher
jour_ordre = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

# Comptage des appels par jour, réindexé dans l'ordre voulu
appels_par_jour = donnees_appels['jour_semaine'].value_counts().reindex(jour_ordre)

# Affichage en texte
print("Nombre d'appels par jour de la semaine :")
print(appels_par_jour)

# Graphique en barres avec une couleur unique
plt.figure(figsize=(8,5))
sns.barplot(x=appels_par_jour.index, y=appels_par_jour.values, color="#D8BFDB")

plt.title("Nombre d'appels par jour de la semaine (Lundi à Vendredi)")
plt.xlabel("Jour de la semaine")
plt.ylabel("Nombre d'appels")
plt.show()
```

Le nombre total d'appels reçus chaque jour, du lundi au vendredi, est calculé et affiché.

Un graphique en barres illustre cette répartition pour visualiser facilement les jours avec plus ou moins d'appels.

❖ Nombre d'appels par type (file d'attente):

```
: # Comptage des appels par queue_name (file d'attente)
calls_by_type = donnees_appels['queue_name'].value_counts().reset_index()
calls_by_type.columns = ['queue_name', 'nb_calls']

# Taille dynamique du graphique
plt.figure(figsize=(max(12, len(calls_by_type) * 0.3), 6))

sns.barplot(
    x="queue_name",
    y="nb_calls",
    hue="queue_name",
    data=calls_by_type,
    palette="bright",
    legend=False
)

plt.xticks(rotation=90, ha='right')
plt.xlabel("Type d'appel (queue_name)")
plt.ylabel("Nombre d'appels")
plt.title("Nombre d'appels par type")
plt.tight_layout()
plt.show()
```

Les appels sont regroupés par type (queue_name), représentant les différentes files d'attente du centre d'appel.

Un graphique en barres permet de comparer visuellement le volume d'appels traité par chaque file.

❖ Top 10 des services les plus demandés:

```
[12]: # Comptage des appels par queue_name les plus demandés (file d'attente)
appels_par_queue = donnees_appels['queue_name'].value_counts()

# Affichage des nombres (top 10)
print("Nombre d'appels par type de file d'attente (top 10) :")
print(appels_par_queue.head(10))

# Graphique en barres pour top 10
plt.figure(figsize=(10,5))
appels_par_queue.head(10).plot(kind='bar', color='mediumseagreen')

plt.title("Top 10 files d'attente les plus utilisées")
plt.ylabel("Nombre d'appels")
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Cette visualisation présente les 10 types de files d'attente (queue_name) ayant reçu le plus grand nombre d'appels.

Un graphique en barres met en évidence les files les plus sollicitées dans le centre d'appel.

❖ Top 10 des services les moins demandés:

```
[14]: # Comptage des appels par queue_name les moins demandés (file d'attente)
top_services_moins = donnees_appels['queue_name'].value_counts().sort_values().head(10)

# Affichage texte
print("\nTop 10 services les moins demandés (queue_name) :")
print(top_services_moins)

# Affichage graphique
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
top_services_moins.plot(kind='bar', color="#FF7F50")
plt.title("Top 10 services les moins demandés (queue_name)")
plt.xlabel("Nom du service (queue_name)")
plt.ylabel("Nombre d'appels")
plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

Cette visualisation montre les 10 types de services (ou files d'attente) ayant reçu le plus faible nombre d'appels.

Elle permet d'identifier les files peu sollicitées dans le centre d'appel.

❖ Top 10 des agents les plus actifs:

```

5]: # Top 10 des agents ayant traité plus d'appels
# Comptage des activités par agent
top_agents = donnees_activites['agent_id'].value_counts().head(10)
print(top_agents)

plt.figure(figsize=(8, 8))
plt.pie(
    top_agents,
    labels=top_agents.index.astype(str),
    autopct='%.1f%%',
    startangle=90,
    colors=plt.cm.tab10.colors
)

plt.title("Répartition des activités parmi les 10 agents les plus actifs")
plt.axis('equal')
plt.tight_layout()
plt.show()

```

Les 10 agents ayant traité le plus d'activités sont identifiés.

Une représentation en camembert illustre leur part respective dans le volume total d'activités, mettant en évidence les agents les plus sollicités.

❖ Top 10 des agents les moins actifs:

```

[16]: # Comptage des appels par agent (agents les moins actifs)
agents_moins_actifs = donnees_activites['agent_id'].value_counts().sort_values()
agents_moins_actifs = agents_moins_actifs[agents_moins_actifs > 0].head(10)

# Affichage texte
print("\Agents les moins actifs (ayant au moins 1 appel) :")
print(agents_moins_actifs)

# Barplot horizontal
plt.figure(figsize=(8, 5))
sns.barplot(
    x=agents_moins_actifs.values,
    y=agents_moins_actifs.index.astype(str),
    palette='Pastel1',
    hue=agents_moins_actifs.index.astype(str),
    legend=False
)
plt.title("Top 10 agents les moins actifs (nombre d'appels)")
plt.xlabel("Nombre d'appels")
plt.ylabel("Agent Number")
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

```

Cette analyse met en évidence les 10 agents ayant traité le plus faible nombre d'activités (au moins une).

Un graphique en barres horizontales permet de visualiser les agents les moins sollicités.

❖ Distribution du nombre d'appels par agent:

```
[17]: # Distribution du nombre d'appels par agent
plt.figure(figsize=(10, 5))
donnees_appels['agent_number'].value_counts().plot(kind='hist', bins=30, color='orchid')
plt.title("Distribution du nombre d'appels par agent")
plt.xlabel("Nombre d'appels")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Cette visualisation montre la répartition du nombre d'appels traités par agent.

L'histogramme permet d'observer la fréquence des agents selon leur charge d'appels, mettant en évidence les écarts d'activité entre agents.

❖ Évolution du nombre d'appels par jour:

```
[18]: # Analyse du nombre d'appels par jour
# Extraire la date sans l'heure
donnees_appels['date'] = donnees_appels['date_received'].dt.date

# Nombre d'appels par jour
plt.figure(figsize=(14,5))
donnees_appels.groupby('date').size().plot()
plt.title("Nombre d'appels par jour")
plt.ylabel("Nombre d'appels")
plt.grid(True)
plt.show()
```

Le graphique affiche le volume quotidien d'appels reçus sur l'ensemble de la période.

Cette courbe permet de visualiser les tendances, pics et variations d'activité au fil du temps.

❖ Volume d'appels par heure et jour de la semaine (heatmap):

```
[19]: # Visualisation du volume d'appels par heure et jour de la semaine (Heatmap)
jour_ordre = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

donnees_appels['date_received'] = pd.to_datetime(donnees_appels['date_received'], errors='coerce')

donnees_appels['heure'] = donnees_appels['date_received'].dt.hour

pivot_jour_heure = donnees_appels.pivot_table(
    index=donnees_appels['date_received'].dt.day_name(),
    columns='heure',
    values='date_received',
    aggfunc='count'
).reindex(jour_ordre)

plt.figure(figsize=(14,6))
sns.heatmap(pivot_jour_heure, cmap="YlGnBu", linewidths=.5, linecolor='gray', annot=True, fmt='g')

plt.title("Volume d'appels par jour de semaine et heure")
plt.ylabel("Jour de la semaine")
plt.xlabel("Heure")
plt.show()
```

Une carte thermique (heatmap) montre le volume d'appels en fonction du jour de la semaine et de l'heure.

Elle permet d'identifier visuellement les périodes de forte et de faible activité dans la semaine.

❖ Évolution mensuelle du volume d'appels:

```
[]: # Évolution mensuelle du volume d'appels reçus
donnees_appels['month'] = donnees_appels['date_received'].dt.to_period("M")

# Grouper par mois et compter le nombre d'appels
calls_by_month = donnees_appels.groupby("month").size()

plt.figure(figsize=(12, 5))
calls_by_month.plot(marker="o", linestyle='-', color='teal')

plt.title("Évolution du nombre d'appels par mois")
plt.xlabel("Mois")
plt.ylabel("Nombre d'appels")
plt.grid(True)
plt.tight_layout()
plt.show()
```



Ce graphique montre le nombre total d'appels reçus chaque mois.

Il permet d'observer les variations d'activité au fil de l'année et d'identifier d'éventuelles tendances saisonnières.

❖ Top 10 des journées les plus chargées:

```
[21]: # Top 10 des journées les plus chargées en appels

top10_jours = donnees_appels['date'].value_counts().sort_values(ascending=False).head(10)

# Affichage en texte
print("\nTop 10 jours avec le plus d'appels :")
print(top10_jours)

# Affichage en graphique
plt.figure(figsize=(10,6))
top10_jours.sort_values().plot(kind='barh', color='coral') # ordre croissant pour barre horizontale

plt.title("Top 10 jours avec le plus d'appels")
plt.xlabel("Nombre d'appels")
plt.ylabel("Date")
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



Ce graphique présente les 10 journées ayant enregistré le plus grand nombre d'appels.

Il met en évidence les pics d'activité exceptionnelles au cours de l'année.

❖ Durée moyenne d'activité par agent (Top 10):

```
: # Calcul durée moyenne par agent (top 10)
duree_moyenne = donnees_activites.groupby('agent_id')['duration'].mean().sort_values(ascending=False).head(10)

# Préparation DataFrame
df_duree_moyenne = duree_moyenne.reset_index()
df_duree_moyenne.columns = ['agent_id', 'duree_moyenne_sec']

plt.figure(figsize=(12,6))
sns.barplot(
    data=df_duree_moyenne,
    x='agent_id',
    y='duree_moyenne_sec',
    hue='agent_id',
    palette='coolwarm',
)
plt.title("Durée moyenne d'activité par agent (top 10)")
plt.xlabel("Agent ID")
plt.ylabel("Durée moyenne (secondes)")
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()

plt.legend([],[], frameon=False)

plt.show()
```

Ce graphique affiche la durée moyenne des activités traitées par les 10 agents les plus actifs.

Il permet d'identifier les agents dont les interventions sont en moyenne les plus longues.

❖ Temps moyen d'attente par heure:

```
: # Temps moyen d'attente par heure
print("\nTemps moyen d'attente par heure :")
moyenne_attente_par_heure = donnees_appels.groupby('hour')['waiting_time'].mean()
print(moyenne_attente_par_heure)

plt.figure(figsize=(12,5))
ax = moyenne_attente_par_heure.plot(kind='bar', color="#4682B4")

plt.title("Temps moyen d'attente selon l'heure")
plt.ylabel("Temps moyen d'attente (secondes)")
plt.xlabel("Heure")
plt.xticks(rotation=45)
plt.grid(axis='y')

for p in ax.patches:
    hauteur = p.get_height()
    ax.annotate(f'{hauteur:.1f}', (p.get_x() + p.get_width() / 2, hauteur),
                ha='center', va='bottom', fontsize=9)

plt.tight_layout()
plt.show()
```

Le graphique présente le temps moyen d'attente des appels pour chaque heure de la journée.

Il permet d'identifier les périodes où les clients attendent le plus longtemps avant d'être pris en charge.

❖ Pourcentage d'appels pris en charge rapidement:

```
[4]: # Pourcentage d'appels pris en charge en moins de 30 secondes, 1 min, 2 min

# Seuils en secondes
seuils = [30, 60, 120]
pourcentages = [(donnees_appels['waiting_time'] <= s).mean() * 100 for s in seuils]

# Graphique barres horizontales
plt.figure(figsize=(8,4))
plt.barrh([f"≤ {s} sec" for s in seuils], pourcentages, color='mediumseagreen')
plt.xlabel("Pourcentage (%)")
plt.title("Pourcentage d'appels pris en charge en moins de x secondes")
plt.xlim(0, 100)
plt.grid(axis='x', linestyle='--', alpha=0.7)

for i, pct in enumerate(pourcentages):
    plt.text(pct + 1, i, f"{pct:.2f} %", va='center')

plt.tight_layout()
plt.show()
```

Ce graphique montre la proportion d'appels traités en moins de 30 secondes, 1 minute et 2 minutes.

Il permet d'évaluer la rapidité de la prise en charge des appels par le centre.

❖ Variabilité des temps d'attente par file d'attente:

```
[25]: # Écart-type des temps d'attente par file d'attente (queue_name)
print("\nÉcart-type des temps d'attente par file d'attente :")
ecart_type_attente = donnees_appels.groupby('queue_name')[['waiting_time']].std().sort_values(ascending=False).head(10)
print(ecart_type_attente)
```

L'écart-type des temps d'attente est calculé pour chaque file d'attente afin de mesurer la variabilité.

Le tableau affiche les 10 files présentant la plus grande dispersion des temps d'attente, ce qui peut indiquer des fluctuations importantes dans la gestion des appels.

❖ Taux de réponses rapides (< 1 minute) par heure

```
[1]: # Taux de réponses rapides (< 1 minute) des appels par heure de la journée
# Création d'une colonne booléenne : 1 si attente <= 60 sec, sinon 0
donnees_appels['reponse_rapide'] = (donnees_appels['waiting_time'] <= 60).astype(int)

# Groupby heure : total appels et total réponses rapides
grouped = donnees_appels.groupby('heure').agg(
    total_appels=('waiting_time', 'count'),
    reponses_rapides=('reponse_rapide', 'sum')
)

# Taux en %
grouped['taux_reponse_rapide'] = (grouped['reponses_rapides'] / grouped['total_appels']) * 100

# Affichage graphique
plt.figure(figsize=(12,6))
sns.lineplot(data=grouped, x=grouped.index, y='taux_reponse_rapide', marker='o', color='mediumseagreen')
plt.axhline(80, color='red', linestyle='--', label='Seuil cible 80%')
plt.title("Taux de réponse rapide (<1 min) par heure")
plt.xlabel("Heure de la journée")
plt.ylabel("Taux de réponse rapide (%)")
plt.xticks(range(8, 20))
plt.ylim(0, 100)
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

Ce graphique montre, heure par heure, le pourcentage d'appels pris en charge en moins d'une minute.

Une ligne rouge indique un seuil cible de 80 % pour évaluer la performance du centre d'appel tout au long de la journée.

❖ Répartition du volume d'appels par agent et file d'attente (Top 10 agents):

```
[27]: # Répartition du volume d'appels par agent et file d'attente (Top 10 agents)
# Calcul de la matrice agent x queue_name
combinaison = donnees_appels.groupby(['agent_number', 'queue_name']).size().unstack(fill_value=0)

# Top 10 agents les plus actifs
top_agents = donnees_activites['agent_id'].value_counts().head(10).index
combinaison_top = combinaison.loc[top_agents]

plt.figure(figsize=(14,7))
sns.heatmap(combinaison_top, cmap="YlOrBr", linewidths=0.5, annot=True, fmt="d")

plt.title("Volume d'appels par agent et file d'attente (Top 10 agents)")
plt.xlabel("File d'attente (queue_name)")
plt.ylabel("Agent")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Cette heatmap présente le nombre d'appels traités par les 10 agents les plus actifs, répartis selon les différentes files d'attente.

Elle permet d'identifier quels agents sont spécialisés ou plus sollicités sur certains types de files.

❖ **Distribution des temps d'attente des appels:**

```
[28]: # Analyse de la distribution des temps d'attente des appels
plt.figure(figsize=(8,4))
sns.boxplot(data=donnees_appels, x='waiting_time', color='lightblue')
plt.title("Distribution globale des temps d'attente")
plt.xlabel("Temps d'attente (secondes)")
plt.grid(True)
plt.show()
```

Le boxplot présente la répartition globale des temps d'attente avant prise en charge des appels. Il met en évidence la médiane, l'étendue et les éventuelles valeurs extrêmes, offrant une vision claire de la variabilité des délais d'attente.

II. Conception et implémentation Java (simulation)

Ce projet a été entièrement réalisé sous IntelliJ IDEA, un environnement de développement intégré (IDE) performant, avec un projet Java structuré via Maven pour assurer une bonne gestion des bibliothèques et de la compilation.

1. Structure du code

Le projet Java a été créé sous forme d'un projet **Maven**, facilitant la gestion des dépendances et la compilation via un fichier pom.xml.

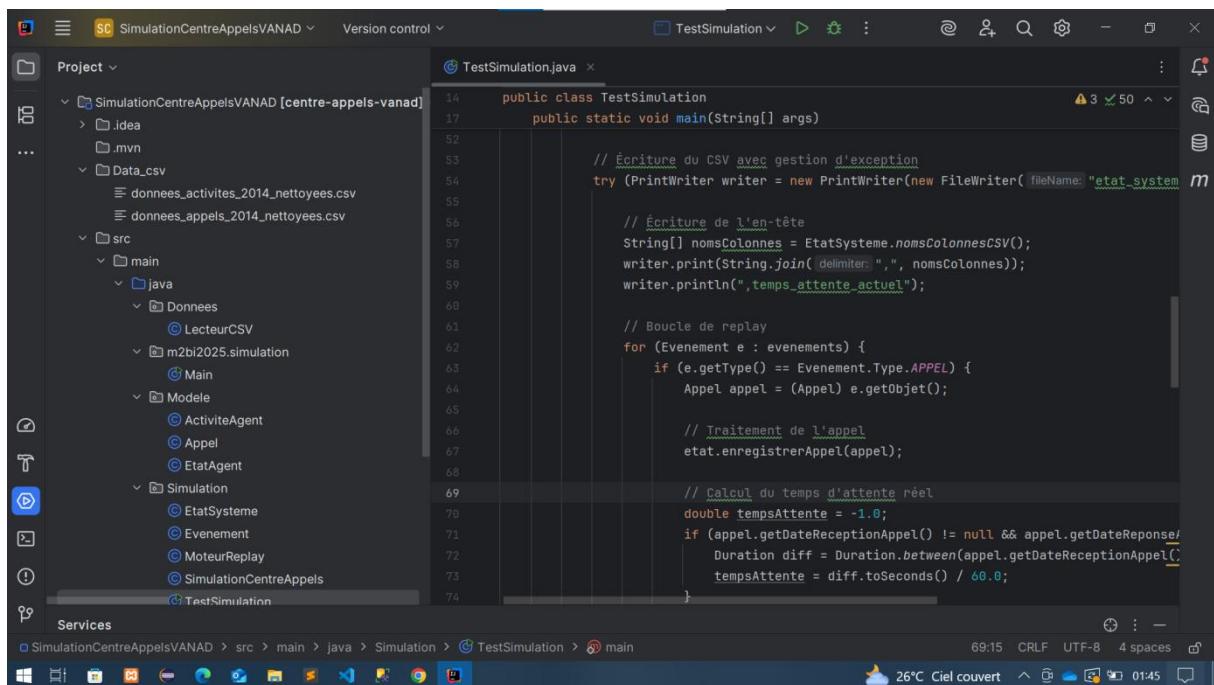
❖ **Les principales dépendances ajoutées sont :**

- ◆ **SSJ** (Stochastic Simulation in Java) version 3.3.2, pour la modélisation et simulation stochastique.
- ◆ **OpenCSV** version 5.11.1, utilisée pour la lecture et le parsing des fichiers CSV.

Le projet est organisé en plusieurs packages pour une meilleure modularité :

❖ **Modele** : regroupe les classes représentant les objets fondamentaux du domaine :

- ◆ **Appel** : modélise un appel téléphonique.
 - ◆ **ActiviteAgent** : représente une activité réalisée par un agent.
 - ◆ **EtatAgent** : décrit l'état d'un agent à un instant donné.
- ❖ **Simulation** : contient les classes gérant la simulation dynamique du centre d'appel :
- ◆ **EtatSystème** : modélise l'état global du système à chaque instant.
 - ◆ **MoteurReplay** : moteur principal assurant la simulation temporelle fidèle.
 - ◆ **TestSimulation** : classe de test pour valider les fonctionnalités.
 - ◆ **SimulationCentreAppels** : orchestrateur global de la simulation.
 - ◆ **Evenement** : classe représentant les événements simulés.
- ❖ **Donnees** : responsable du traitement des fichiers CSV :
- ◆ **LecteurCSV** : lecture et parsing des fichiers d'appels et d'activités.
- ❖ Un dossier **data_csv** contient les fichiers de données nettoyées (**donnees_appels_2014_nettoyees.csv** et **donnees_activites_2014_nettoyees.csv**) utilisés pour la simulation.



The screenshot shows the IntelliJ IDEA interface with the following details:

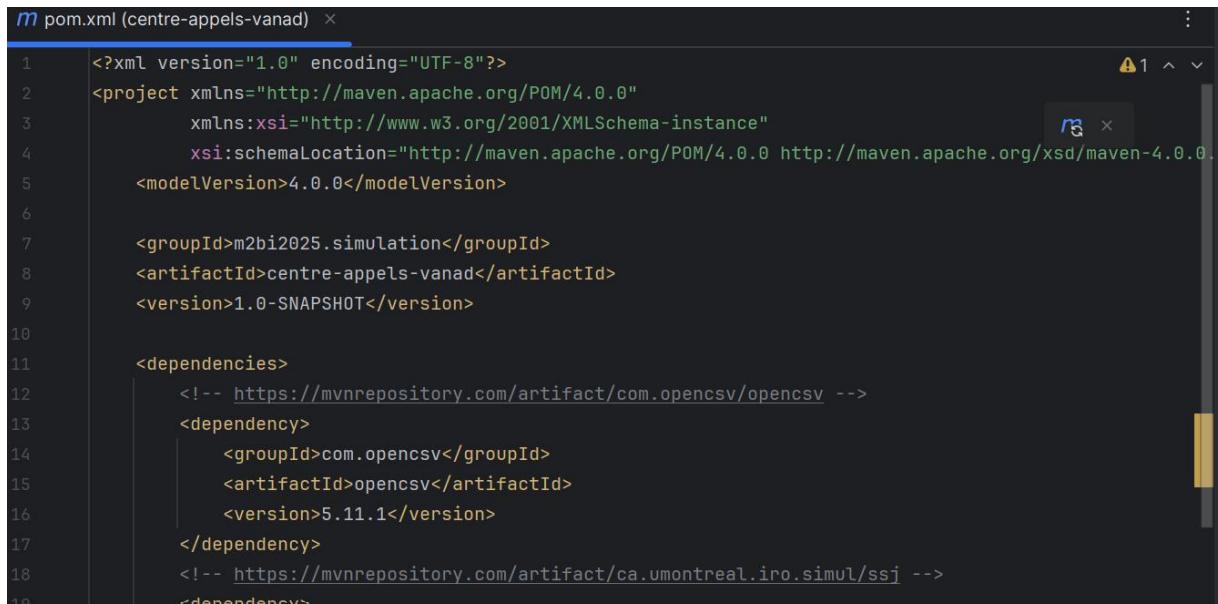
- Project View:** Shows the project structure under "SimulationCentreAppelsVANAD". It includes:
 - src folder containing main and java subfolders.
 - java folder containing Donnees, LecteurCSV, m2bi2025.simulation, Modelé, and Simulation subfolders.
 - SimulationCentreAppelsVANAD folder containing .idea and .mvn subfolders.
 - Data_csv folder containing donnees_activites_2014_nettoyees.csv and donnees_appels_2014_nettoyees.csv.
- Code Editor:** Displays the `TestSimulation.java` file with the following content:

```

14  public class TestSimulation
15      public static void main(String[] args)
16
17          // Écriture du CSV avec gestion d'exception
18          try (PrintWriter writer = new PrintWriter(new FileWriter("etat_systeme.csv")))
19          {
20              // Écriture de l'en-tête
21              String[] nomsColonnes = EtatSystème.nomsColonnesCSV();
22              writer.print(String.join(", ", nomsColonnes));
23              writer.println("temps_attente_actuel");
24
25              // Boucle de replay
26              for (Evenement e : evenements) {
27                  if (e.getType() == Evenement.Type.APPEL) {
28                      Appel appel = (Appel) e.getObjet();
29
30                      // Traitement de l'appel
31                      etat.enregistrerAppel(appel);
32
33                      // Calcul du temps d'attente réel
34                      double tempsAttente = -1.0;
35                      if (appel.getDateReceptionAppel() != null &amp; appel.getDateReponse() != null)
36                          Duration diff = Duration.between(appel.getDateReceptionAppel(),
37                                                          appel.getDateReponse());
38                      tempsAttente = diff.toSeconds() / 60.0;
39                  }
40              }
41          }
42      }
43  
```

2. Explication des classes:

❖ Configuration du projet Maven



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>m2bi2025.simulation</groupId>
    <artifactId>centre-appels-vanad</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/com.opencsv/opencsv -->
        <dependency>
            <groupId>com.opencsv</groupId>
            <artifactId>opencsv</artifactId>
            <version>5.11.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/ca.umontreal.iro.simul:ssj -->
        <dependency>
```

Le fichier **pom.xml** (Project Object Model) est le cœur de la configuration d'un projet Java utilisant l'outil de gestion de projet **Maven**. Il sert à :

- ◆ **Définir l'identité du projet** (nom, version, organisation),
- ◆ **Lister les bibliothèques nécessaires** (appelées *dépendances*),
- ◆ **Configurer la compilation et l'encodage** (par exemple, version de Java utilisée).

➤ Rôles et fonctions principales :

◆ **Identification du projet :**

Il précise le nom du projet, son identifiant unique, ainsi que sa version. Cela permet de l'organiser correctement dans un environnement Maven.

◆ **Gestion automatique des bibliothèques :**

Il déclare les bibliothèques externes dont le projet dépend. Par exemple :

- ❖ **OpenCSV** pour la lecture des fichiers CSV (utilisée dans la classe LecteurCSV),

- ❖ **SSJ** pour la simulation stochastique.

Grâce à Maven, ces bibliothèques sont automatiquement téléchargées et mises à jour depuis un dépôt central.

➤ **Paramétrage du compilateur Java :**

Il fixe la version de Java à utiliser (ici Java 17), ainsi que l'encodage des fichiers (UTF-8), garantissant la compatibilité du code sur toutes les machines.

➤ **Utilisation dans le projet VANAD :**

Le fichier pom.xml permet :

- ◆ D'exécuter le projet sans installer manuellement les bibliothèques nécessaires,
- ◆ D'assurer une compilation correcte et cohérente du projet,
- ◆ De faciliter la reproductibilité de l'environnement de développement pour tous les membres du groupe.

❖ **Modele:**

La classe **Appel** représente un **appel téléphonique** enregistré dans le système VANAD. Elle modélise les différentes étapes de vie d'un appel, ainsi que les informations temporelles nécessaires à l'analyse ou à la simulation.

```
15  /*
16   * public class Appel 30 usages
17   *
18   *     // === Données principales de l'appel ===
19   *     private LocalDateTime dateReceptionAppel; 3 usages
20   *     private String nomFileAttenteClient; 5 usages
21   *     private Integer identifiantAgent; 5 usages
22   *
23   *     // === Événements associés à l'appel ===
24   *     private LocalDateTime dateReponseAgent; 4 usages
25   *     private LocalDateTime dateConsultation; 4 usages
26   *     private LocalDateTime dateTransfert; 4 usages
27   *     private LocalDateTime dateRaccrochage; 4 usages
28   *
29   *     private Integer anneeAppel; 3 usages
30   *     private Integer moisAppel; 3 usages
31   *     private Integer jourAppel; 3 usages
32   *     private Integer numeroJourSemaine; 3 usages
33   *     private Integer heureAppel; 4 usages
34   *     private Integer minuteAppel; 4 usages
35   *     private Double tempsDansJournee; 3 usages
36   *
37   *     // Format utilisé uniquement pour le constructeur CSV brut
38   */
```

➤ Rôles principaux :

- ◆ Représenter un appel avec ses **données chronologiques** : réception, réponse, consultation, transfert, fin.
- ◆ Associer l'appel à une **file d'attente** (queue_name) et éventuellement à un **agent** (si connu).
- ◆ Fournir des **variables dérivées temporelles** (heure, jour, etc.) utiles pour les prédictions ou les regroupements statistiques.

➤ Fonctionnalités clés :

- ◆ **Constructeur depuis CSV brut** : permet de lire automatiquement une ligne CSV contenant les informations d'un appel (avec gestion sécurisée des erreurs de format).
- ◆ **Mise à jour automatique des champs dérivés** lors de l'appel à setDateReceptionAppel(...) (extraction de l'année, jour de la semaine, heure, etc.).
- ◆ **Accesseurs (getters/setters)** : tous les champs sont accessibles et modifiables individuellement pour une flexibilité maximale.

- ◆ **Méthode** `toString()` : fournit une représentation textuelle complète de l'appel (utile pour le débogage ou l'export console).

➤ **Exemple de champs modélisés :**

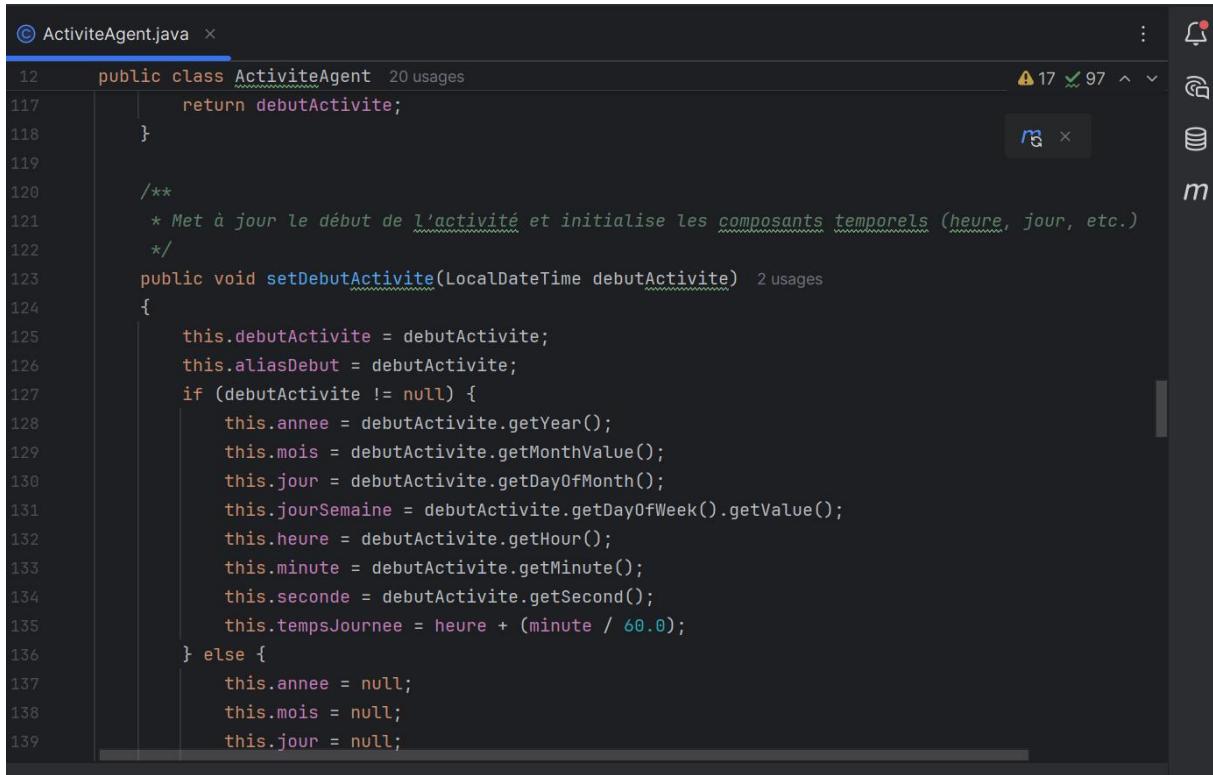
- ◆ `dateReceptionAppel, dateReponseAgent, dateConsultation, dateTransfert,`
`dateRaccrochage`
- ◆ `nomFileAttenteClient, identifiantAgent`
- ◆ Champs dérivés : `heureAppel, numeroJourSemaine, tempsDansJournee...`

➤ **Utilisation dans le projet :**

Cette classe est utilisée par :

- ◆ LecteurCSV pour créer les appels depuis les fichiers CSV,
- ◆ MoteurReplay pour la simulation temporelle,
- ◆ EtatSysteme pour extraire les variables du vecteur d'entrée ANN.

La classe **ActiviteAgent** représente une **activité réalisée par un agent** dans le centre d'appel VANAD. Elle modélise les périodes de travail ou d'indisponibilité des agents à partir des données extraites du fichier CSV des activités.



```
12  public class ActiviteAgent { 20 usages
117     return debutActivite;
118 }
119
120 /**
121 * Met à jour le début de l'activité et initialise les composants temporels (heure, jour, etc.)
122 */
123 public void setDebutActivite(LocalDateTime debutActivite) { 2 usages
124
125     this.debutActivite = debutActivite;
126     this.aliasDebut = debutActivite;
127     if (debutActivite != null) {
128         this.annee = debutActivite.getYear();
129         this.mois = debutActivite.getMonthValue();
130         this.jour = debutActivite.getDayOfMonth();
131         this.jourSemaine = debutActivite.getDayOfWeek().getValue();
132         this.heure = debutActivite.getHour();
133         this.minute = debutActivite.getMinute();
134         this.seconde = debutActivite.getSecond();
135         this.tempsJournee = heure + (minute / 60.0);
136     } else {
137         this.annee = null;
138         this.mois = null;
139         this.jour = null;

```

➤ Rôles principaux :

- ◆ Représenter les **plages de disponibilité ou d'activité** d'un agent à travers des dates de début et de fin.
- ◆ Fournir des **informations temporelles dérivées** (jour, heure, durée, etc.) utiles pour la simulation et l'analyse.
- ◆ Permettre d'identifier l'agent et le contexte de son activité (campagne, DND, appel, etc.).

➤ Fonctionnalités clés :

- ◆ **Initialisation depuis un CSV** : tous les attributs (identifiants et timestamps) peuvent être chargés et interprétés depuis les données brutes.
- ◆ **Décomposition temporelle automatique** (année, mois, heure, etc.) à partir de debutActivite.
- ◆ **Calcul automatique de la durée** (en minutes) entre le début et la fin de l'activité.

- ◆ **Alias** (aliasDebut, aliasFin, aliasAgent) utilisés pour faciliter la compatibilité avec d'autres classes du système.
- ◆ **Méthode** `toString()` : utile pour le débogage et l'affichage synthétique de l'activité.

➤ **Attributs clés :**

- ◆ Données métier : idActivite, idAgent, debutActivite, finActivite, dureeMinutes
- ◆ Dérivées temporelles : heure, jourSemaine, tempsJournee, etc.
- ◆ Identifiants secondaires : idUtilisateur, idCampagne, idDnd, idDernierAppel, extension

Cette classe est utilisée par :

- ◆ LecteurCSV pour lire les activités depuis les fichiers VANAD,
- ◆ MoteurReplay pour connaître les périodes de présence des agents dans la simulation,
- ◆ EtatSysteme pour enrichir le contexte dynamique au moment d'un appel.

La classe **EtatAgent** modélise l'état de disponibilité d'un agent à un instant donné. Elle est utilisée par le moteur de simulation (MoteurReplay) pour déterminer si un agent est éligible et disponible pour prendre en charge un appel, en tenant compte de ses périodes de travail, d'occupation et des services qu'il peut traiter.

```
© EtatAgent.java ×

14  public class EtatAgent  5 usages
90      public void setIndispoAvant(LocalDateTime indispoAvant)  1 usage
91      {
92          this.indispoAvant = indispoAvant;
93      }
94
95      public void setOccupeJusquA(LocalDateTime occupeJusquA)  1 usage
96      {
97          this.occupeJusquA = occupeJusquA;
98      }
99
100     // === Affichage lisible pour le débogage ===
101
102     @Override
103     public String toString() {
104         return "StatutAgent{" +
105             "servicesAutorises=" + servicesAutorises +
106             ", indispoAvant=" + indispoAvant +
107             ", occupeJusquA=" + occupeJusquA +
108             '}';
109     }
110 }
```

➤ Rôles principaux :

Gérer les **restrictions de service** : chaque agent peut traiter uniquement certains types d'appels (servicesAutorises).

Contrôler la **disponibilité temporelle** de l'agent : via trois horodatages clés :

- ◆ dispoApres : instant à partir duquel l'agent est disponible.
- ◆ indispoAvant : début éventuel d'une période d'indisponibilité.
- ◆ occupeJusquA : instant jusqu'auquel l'agent est occupé.

➤ Fonctionnalités principales :

- ◆ Méthode estDisponible(moment) : vérifie si un agent peut être affecté à un appel à un instant moment donné.
- ◆ Méthode accepteService(nomService) : vérifie si l'agent est autorisé à traiter un certain service ou file d'attente.

- ◆ Fournit une **vision dynamique** et actualisée de l'état opérationnel d'un agent à tout moment de la simulation.

➤ **Attributs clés :**

- ◆ servicesAutorises : ensemble des noms de files d'attente que l'agent peut gérer.
- ◆ dispoApres, indispoAvant, occupeJusquA : indicateurs temporels de disponibilité.

➤ **Utilisation dans le projet :**

- ◆ Cette classe est **instanciée pour chaque agent** dans la simulation.
- ◆ Elle est mise à jour à chaque **nouvel appel ou activité** afin de refléter les changements de statut de l'agent.
- ◆ Elle permet à la classe EtatSysteme ou MoteurReplay de prendre des **décisions d'affectation intelligentes**.

❖ **Données:**

La classe LecteurCSV est une classe utilitaire chargée de lire les fichiers de données historiques au format CSV fournis par le centre d'appel VANAD. Elle transforme ces fichiers bruts en objets Java exploitables dans la simulation (Appel et ActiviteAgent).

```
④ LecteurCSV.java x
17 public class LecteurCSV 6 usages
18     // Format standard utilisé pour parser les dates présentes dans les CSV
19     private static final DateTimeFormatter FORMAT_DATE = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")
20
21
22
23 /**
24 * Lit un fichier CSV contenant les appels clients et retourne une liste d'objets Appel.
25 */
26 @
27 public static List<Appel> lireAppels(String cheminFichier) throws Exception 2 usages
28 {
29     List<Appel> appels = new ArrayList<>();
30     try (CSVReader lecteur = new CSVReader(new FileReader(cheminFichier))) {
31         String[] ligne;
32         int numeroLigne = 1;
33         lecteur.readNext();
34
35         while ((ligne = lecteur.readNext()) != null) {
36             numeroLigne++;
37             try {
38                 Appel appel = new Appel();
39
40                     appel.setDateReceptionAppel(parseDateSecurisee(ligne, index: 0, nomChamp: "date_reception"));
41                     appel.setNomFileAttenteClient(parseChaineSecurisee(ligne, index: 1));
42                     appel.setIdentifiantAgent(parseNombreEntierFlexible(ligne, index: 2, nomChamp: "agent_id"));
43
44             }
45         }
46     }
47 }
```

Rôles principaux :

- ◆ Lire les fichiers CSV contenant les **appels clients** et les **activités des agents**.
 - ◆ Convertir chaque ligne du fichier en objet Java (instanciation des classes Appel ou ActiviteAgent).
 - ◆ **Filtrer, parser et nettoyer les données** mal formatées ou incomplètes, tout en assurant la robustesse de la lecture.

➤ Fonctionnalités principales :

- ◆ lireAppels(cheminFichier) :
 - ✧ Lit chaque ligne du fichier CSV des appels.
 - ✧ Extrait les dates (réception, réponse, consultation, transfert, raccrochage).
 - ✧ Corrige les identifiants d'agent mal formatés (ex. 8391.0 au lieu de 8391).
 - ✧ Retourne une liste d'objets Appel.
 - ◆ lireActivites(cheminFichier) :

- ❖ Lit les activités des agents (changements de statut).
- ❖ Parse les identifiants, extensions, dates de début et fin d’activité.
- ❖ Retourne une liste d’objets ActiviteAgent.

➤ **Gestion des erreurs :**

- ◆ Détection automatique des champs manquants ou mal formatés.
- ◆ Affichage de messages d’erreur clairs avec indication de la ligne fautive.
- ◆ Tolérance aux champs nulls ou facultatifs.

➤ **Méthodes utilitaires internes :**

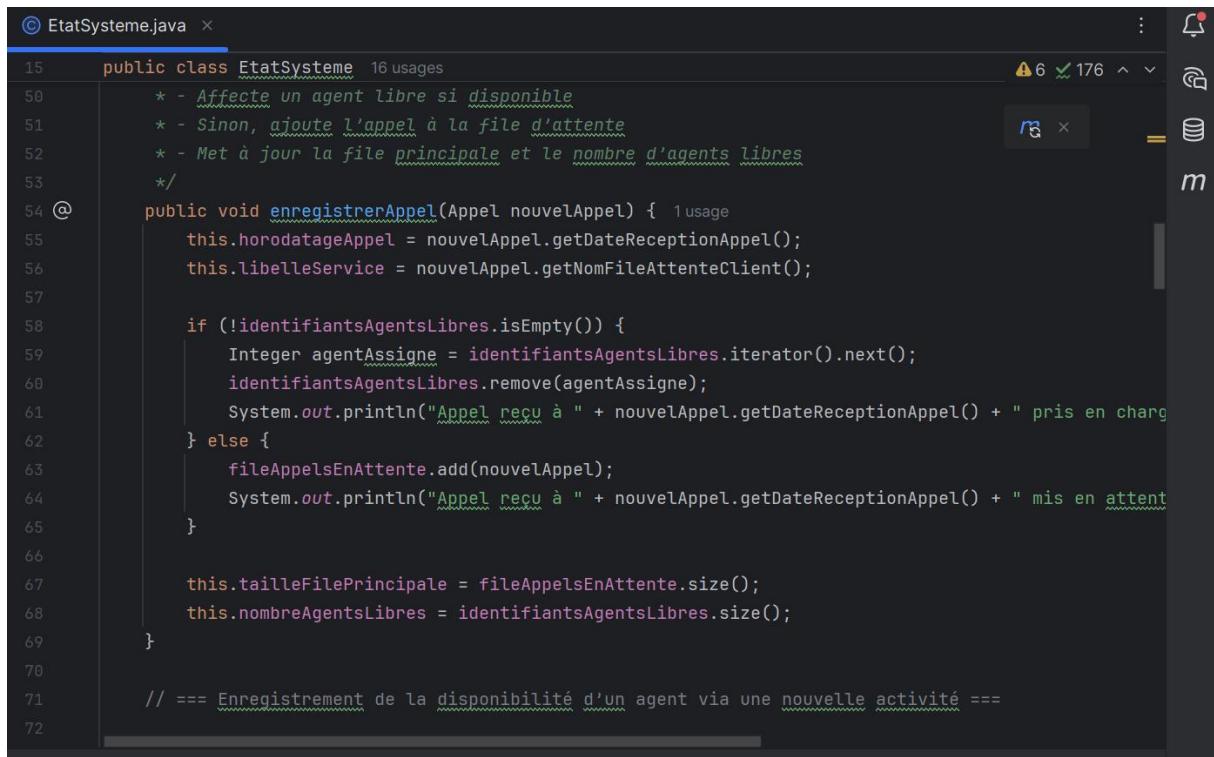
- ◆ parseDateSecurisee : Convertit les dates en LocalDateTime, avec ou sans tolérance au champ vide.
- ◆ parseEntierSecurise / parseLongSecurise : Conversion sécurisée des entiers, avec gestion des erreurs.
- ◆ parseNombreEntierFlexible : Gère les entiers écrits comme des décimaux (ex. 1234.0).
- ◆ parseChaineSecurisee : Nettoie les chaînes de texte, retourne null si vide.

➤ **Utilisation dans le projet :**

- ◆ Cette classe est utilisée en amont de toute simulation pour **préparer les données d’entrée**.
- ◆ Elle garantit que les objets manipulés dans les classes de simulation (SimulationCentreAppels, TestSimulation, MoteurReplay) sont **valides, cohérents et complets**.
- ◆ Elle constitue une **interface robuste entre les fichiers CSV bruts** et les composants du moteur de simulation.

❖ Simulation:

La classe **EtatSystème** joue un rôle **central dans la simulation** du centre d'appel. Elle représente à chaque instant l'état global du système, à la fois pour le **pilotage de la simulation** et pour la **génération de données d'entraînement** destinées aux modèles prédictifs (LES, Avg-LES, ANN).



```
© EtatSystème.java x
15  public class EtatSystème { 16 usages
16      * - Affecte un agent libre si disponible
17      * - Sinon, ajoute l'appel à la file d'attente
18      * - Met à jour la file principale et le nombre d'agents libres
19      */
20
21     @
22     public void enregistrerAppel(Appel nouvelAppel) { 1 usage
23         this.horodateAppel = nouvelAppel.getDateReceptionAppel();
24         this.libelleService = nouvelAppel.getNomFileAttenteClient();
25
26         if (!identifiantsAgentsLibres.isEmpty()) {
27             Integer agentAssigne = identifiantsAgentsLibres.iterator().next();
28             identifiantsAgentsLibres.remove(agentAssigne);
29             System.out.println("Appel reçu à " + nouvelAppel.getDateReceptionAppel() + " pris en charge");
30         } else {
31             fileAppelsEnAttente.add(nouvelAppel);
32             System.out.println("Appel reçu à " + nouvelAppel.getDateReceptionAppel() + " mis en attente");
33         }
34
35         this.tailleFilePrincipale = fileAppelsEnAttente.size();
36         this.nombreAgentsLibres = identifiantsAgentsLibres.size();
37     }
38
39     // === Enregistrement de la disponibilité d'un agent via une nouvelle activité ===
40 }
```

➤ Rôles principaux :

- ◆ **Suivre dynamiquement l'état du système** : agents disponibles, file d'attente, répartition des appels.
- ◆ **Enregistrer les événements** : nouveaux appels, nouvelles activités d'agents.
- ◆ **Transformer l'état courant en vecteur de caractéristiques** pour alimenter un modèle supervisé.
- ◆ **Calculer et stocker des prédicteurs** comme LES et Avg-LES pour comparaison.

➤ Fonctions principales :

- ◆ enregistrerAppel(Appel) :
 - ✧ Simule la réception d'un appel.
 - ✧ Affecte un agent libre si disponible, sinon ajoute l'appel à la file d'attente.
 - ✧ Met à jour la taille de la file et le nombre d'agents libres.
- ◆ enregistrerActiviteAgent(ActiviteAgent) :
 - ✧ Signale la disponibilité d'un agent après une activité.
 - ✧ Si des appels sont en attente, l'agent libre est immédiatement affecté à l'un d'eux.
- ◆ transformerEnVecteurCaracteristiques() :
 - ✧ Encode l'état du système (service, heure, jour, files d'attente, agents disponibles, etc.) sous forme de tableau numérique (vecteur de 13 dimensions).
 - ✧ Utilisé pour entraîner un modèle de prédiction du temps d'attente.
- ◆ nomsColonnesCSV() :
 - ✧ Fournit les noms des colonnes utilisées dans le fichier CSV d'entraînement exporté (ex: etat_systeme_sortie.csv).

➤ **Données modélisées :**

- ◆ identifiantsAgentsLibres : ensemble des agents disponibles à un instant donné.
- ◆ fileAppelsEnAttente : file FIFO contenant les appels non encore traités.
- ◆ libelleService, horodatageAppel : informations du dernier appel enregistré.
- ◆ nombreAgentsLibres, tailleFilePrincipale, taillesFilesAnnexes : état de la capacité opérationnelle.
- ◆ estimationLES, estimationLESMoyenne : prédictions internes calculées à chaque appel.

➤ **Importance dans le projet :**

- ◆ Elle est **appelée à chaque itération** de la simulation, lors d'un événement (appel ou activité).
- ◆ Elle agit comme un **collecteur de contexte instantané**, essentiel pour produire des jeux de données temporellement cohérents.
- ◆ Elle constitue la **passerelle entre la simulation Java et l'apprentissage supervisé** en Python.

La classe **MoteurReplay** est le cœur du moteur de simulation temporelle fidèle du centre d'appel VANAD. Elle permet de **rejouer** les événements passés (appels et activités d'agents) dans l'ordre chronologique pour **reconstruire l'état du système à chaque instant**.

Elle est utilisée pour générer des **snapshots du système** (objets EtatSystème) à des moments stratégiques (arrivée d'un appel), en vue d'entraîner des modèles de prédiction du temps d'attente.

```

 17  public class MoteurReplay  2 usages
185  private void collecterStatistiques(Appel appel, String file)  1 usage
193  if (appel.getDateReponseAgent() != null && appel.getDateRaccrochage() != null)
194  {
195      double service = ChronoUnit.SECONDS.between(appel.getDateReponseAgent(), appel.getDateRaccrochage());
196      enregistrerDansHistorique(historiquesTempsService.get(file), service, limite: 200);
197  }
198  }
199
200 // === Ajoute une valeur à l'historique, avec limite de taille ===
201 @
202 private void enregistrerDansHistorique(List<Double> liste, double valeur, int limite)  2 usages
203 {
204     liste.add(valeur);
205     if (liste.size() > limite) liste.remove(index: 0);
206 }
207
208 // === Moyenne d'une liste de valeurs, ou valeur par défaut si vide ===
209 private double calculerMoyenne(List<Double> liste, double valeurParDefaut)  2 usages
210 {
211     return (liste == null || liste.isEmpty())
212         ? valeurParDefaut
213         : liste.stream().mapToDouble(Double::doubleValue).average().orElse(valeurParDefaut);
214 }

```

➤ Objectifs principaux de MoteurReplay :

- ◆ **Reconstituer le fonctionnement du centre d'appel** à partir de données historiques.
- ◆ **Mettre à jour dynamiquement** l'état des agents et des files d'attente.
- ◆ **Capturer les vecteurs de caractéristiques** au moment de chaque appel (pour entraînement ANN).
- ◆ **Calculer les prédicteurs de délai** (LES, Avg-LES) pour chaque appel.

➤ **Structure interne :**

◆ **Données principales :**

- ◊ filesParService : files d'attente distinctes pour chaque service (ex : 30172, 30175...).
- ◊ etatsParAgent : état de chaque agent (disponibilité, compétences...).
- ◊ historiquesTempsAttente, historiquesTempsService : liste des dernières durées pour le calcul de moyennes mobiles.
- ◊ activitesChronologiques : liste triée des activités d'agents (indiquant disponibilité ou indisponibilité).
- ◊ typesServices : services traités (donnée d'entrée).

Principales méthodes :

➤ **capturerEtatSysteme(Appel appel, LocalDateTime horodatage)**

- ◆ Capture l'état du système juste avant le traitement d'un appel :
- ◆ Met à jour les agents disponibles via mettreAJourEtatsAgents.
- ◆ Supprime les appels déjà pris en charge via purgerAppelsAnciennementTraites.
- ◆ Calcule les tailles de file (principale et annexes).
- ◆ Compte les agents compatibles disponibles.
- ◆ Calcule les prédicteurs de délai (LES et Avg-LES).
- ◆ Retourne un objet EtatSysteme.

➤ **enregistrerEvenementAppel(Appel appel)**

- ◆ Insère un appel dans la bonne file et met à jour l'occupation de l'agent s'il a répondu.

Utilisé pour construire dynamiquement la simulation à partir des données brutes.

➤ **mettreAJourEtatsAgents(LocalDateTime maintenant)**

- ◆ Applique les activités d'agents jusqu'à l'instant maintenant :
- ◆ Si l'activité signale une disponibilité (code 3 ou 16), on la note dans dispoApres.
- ◆ Si elle signale une indisponibilité (code 2, 7, etc.), on la note dans indispoAvant.

➤ **calculerPredicteurs(EtatSystème etat, String file)**

- ◆ Calcule les estimations de délai d'attente avec :
- ◆ **LES** : basé sur la position dans la file et la durée moyenne de service.
- ◆ **Avg-LES** : basé sur les attentes précédentes, ajusté selon la charge.

➤ **collecterStatistiques(Appel appel, String file)**

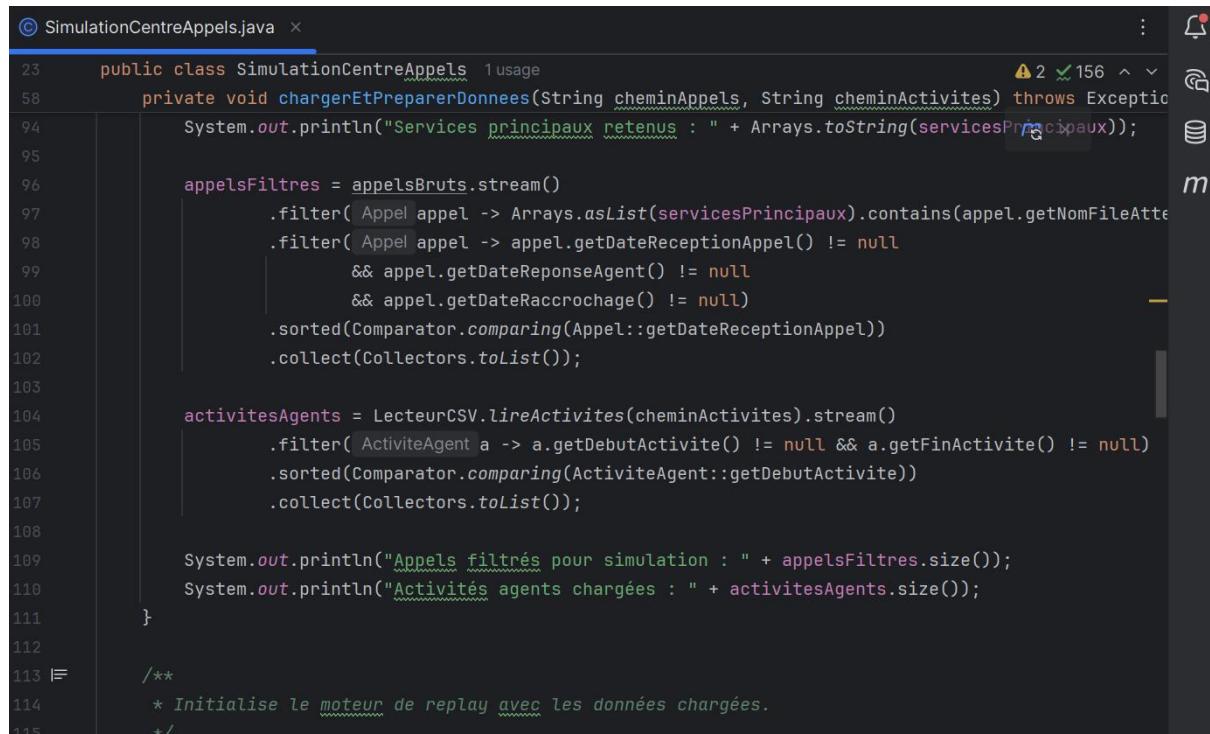
- ◆ Stocke dans l'historique : Le temps d'attente réel (réponse - réception).
- ◆ Le temps de service (raccrochage - réponse).

Ces valeurs servent à alimenter les moyennes pour LES et Avg-LES.

➤ **Autres méthodes :**

- ◆ ajusterTaille(...) : limite le nombre de files annexes à 5.
- ◆ compterAgentsCompatibles(...) : compte les agents aptes à répondre à un service donné à un moment donné.
- ◆ calculerMoyenne(...), enregistrerDansHistorique(...) : outils pour gérer les moyennes glissantes.

La classe **SimulationCentreAppels** joue le rôle central d'orchestrateur. Elle coordonne toutes les étapes de la simulation du centre d'appel VANAD à partir des données historiques réelles. Elle s'appuie sur le moteur MoteurReplay pour rejouer les événements de manière chronologique et générer des données d'entraînement supervisé pour des modèles de prédiction (ANN).



```

 23  public class SimulationCentreAppels 1 usage
 24      private void chargerEtPreparerDonnees(String cheminAppels, String cheminActivites) throws Exception
 25          System.out.println("Services principaux retenus : " + Arrays.toString(servicesPrincipaux));
 26
 27          appelsFiltres = appelsBruts.stream()
 28              .filter( Appel appel -> Arrays.asList(servicesPrincipaux).contains(appel.getNomFichierAppel())
 29                  .filter( Appel appel -> appel.getDateReceptionAppel() != null
 30                      && appel.getDateReponseAgent() != null
 31                      && appel.getDateRaccrochage() != null)
 32              .sorted(Comparator.comparing(Appel::getDateReceptionAppel))
 33              .collect(Collectors.toList());
 34
 35
 36          activitesAgents = LecteurCSV.lireActivites(cheminActivites).stream()
 37              .filter( ActiviteAgent a -> a.getDebutActivite() != null && a.getFinActivite() != null)
 38              .sorted(Comparator.comparing(ActiviteAgent::getDebutActivite))
 39              .collect(Collectors.toList());
 40
 41
 42          System.out.println("Appels filtrés pour simulation : " + appelsFiltres.size());
 43          System.out.println("Activités agents chargées : " + activitesAgents.size());
 44      }
 45
 46
 47      /**
 48      * Initialise le moteur de replay avec les données chargées.
 49      */

```

➤ Objectifs de la classe :

- ◆ Charger et prétraiter les fichiers CSV d'appels et d'activités.
- ◆ Filtrer et trier les événements utiles à la simulation.
- ◆ Initialiser un moteur de simulation temporelle (MoteurReplay).
- ◆ Simuler chaque appel, capture d'état + enregistrement.
- ◆ Exporter les données d'apprentissage vers un fichier CSV.
- ◆ Évaluer la performance des prédicteurs LES et Avg-LES.

➤ Attributs principaux :

- ◆ servicesPrincipaux : les services (queues) les plus fréquents, sélectionnés pour la simulation.
- ◆ appelsFiltres : la liste des appels conservés après filtrage (jours/heures/services).
- ◆ activitesAgents : les périodes de disponibilité ou d'indisponibilité des agents.
- ◆ donneesEntrainement : liste des objets EtatSysteme, constituant le jeu de données final.
- ◆ moteurReplay : instance du moteur MoteurReplay, utilisé pour reconstruire dynamiquement l'état du centre d'appel.

➤ **Méthodes principales :**

◆ **lancerReplayHistorique(...)**

- ✧ Point d'entrée de la simulation. Elle lance les **5 étapes clés** :
- ✧ Chargement des fichiers.
- ✧ Préparation des données (filtrage, tri, sélection des services).
- ✧ Initialisation du moteur de simulation.
- ✧ Exécution du replay historique **événement par événement**.
- ✧ Export des données d'entraînement et **analyse finale**.

◆ **chargerEtPreparerDonnees(...)**

Cette méthode prépare les données à utiliser :

- ✧ **Chargement CSV** via LecteurCSV.
- ✧ **Filtrage temporel** : seuls les appels en jours ouvrés (lundi–vendredi) et dans les heures 8h–20h sont conservés.
- ✧ **Sélection des services principaux** : au moins 200 appels valides chacun, les 5 services les plus représentés.
- ✧ **Nettoyage des appels** : on ne garde que ceux avec des dates valides (réception, réponse, raccrochage).

- ✧ **Tri** chronologique.

- ◆ **initialiserMoteurReplay()**

- ✧ Instancie la classe MoteurReplay avec les appels filtrés, les activités et les services retenus.

- ◆ **executerReplayEvenementParEvenement()**

Simule le **déroulement de chaque appel**, dans l'ordre chronologique :

À chaque appel :

- ✧ Capture un EtatSystème (snapshot au moment de l'appel).
- ✧ Vérifie s'il est **valide pour l'apprentissage**.
- ✧ Enregistre cet état si valide.
- ✧ Met à jour le moteur avec l'appel traité.

- ◆ L'état est gardé seulement si :

- ✧ Le délai observé est réaliste ($0 \leq \text{délai} < 7200$ secondes).
- ✧ Il y a au moins 1 agent libre.
- ✧ La file d'attente principale n'est pas trop longue (< 500).

- ◆ **exporterDonneesEntrainement()**

- ✧ Écrit le fichier jeu_donnees_ann_vanad.csv contenant :
- ✧ Les **13 caractéristiques** extraites de l'état du système (input ANN).
- ✧ Le **temps d'attente réel** (output ANN).

Chaque ligne = un appel, chaque colonne = une variable prédictive.

- ◆ **analyserResultatsSimulation()**

Affiche un **résumé** des échantillons générés :

- ✧ Nombre total d'exemples.
- ✧ Temps d'attente moyen.
- ✧ Taille moyenne des files.

Et appelle la validation des prédicteurs.

◆ **validerPredicteurs()**

- ✧ Calcule et affiche la **RMSE** et **RRMSE** des prédicteurs :
- ✧ **LES (Position × Durée / nb agents).**
- ✧ **Avg-LES (attente moyenne historique × facteur de charge).**

Ce test donne une **référence comparative** avant d'entraîner l'ANN

La classe **Evenement** est une **classe utilitaire essentielle** dans une simulation temporelle réaliste comme celle du centre d'appel VANAD. Elle permet de représenter **tout changement d'état dans le système** que ce soit l'arrivée d'un **appel client** ou une **activité d'un agent** et de les **ordonner chronologiquement** pour les traiter dans l'ordre exact où ils se sont produits dans l'historique.

```

8  * Ces objets sont insérés dans une file de priorité (ordonnée par date) pour simuler
9  */
10 public class Evenement implements Comparable<Evenement> 10 usages
11 {
12     /**
13      * Enumération des types d'événements possibles :
14      * - APPEL : représente l'arrivée d'un nouvel appel
15      * - ACTIVITE : représente le début d'une activité d'agent
16      */
17     public enum Type { APPEL, ACTIVITE } 7 usages
18
19     private Type type; 2 usages
20     private LocalDateTime date; 4 usages
21     private Object objet; // Appel ou ActiviteAgent 2 usages
22
23
24     /**
25      * Constructeur de la classe Evenement.
26      *
27      * @param type Type d'événement (APPEL ou ACTIVITE)
28      * @param date Date à laquelle se produit l'événement
29      * @param objet L'objet associé à l'événement (Appel ou ActiviteAgent)
30      */
31     public Evenement(Type type, LocalDateTime date, Object objet) 2 usages

```

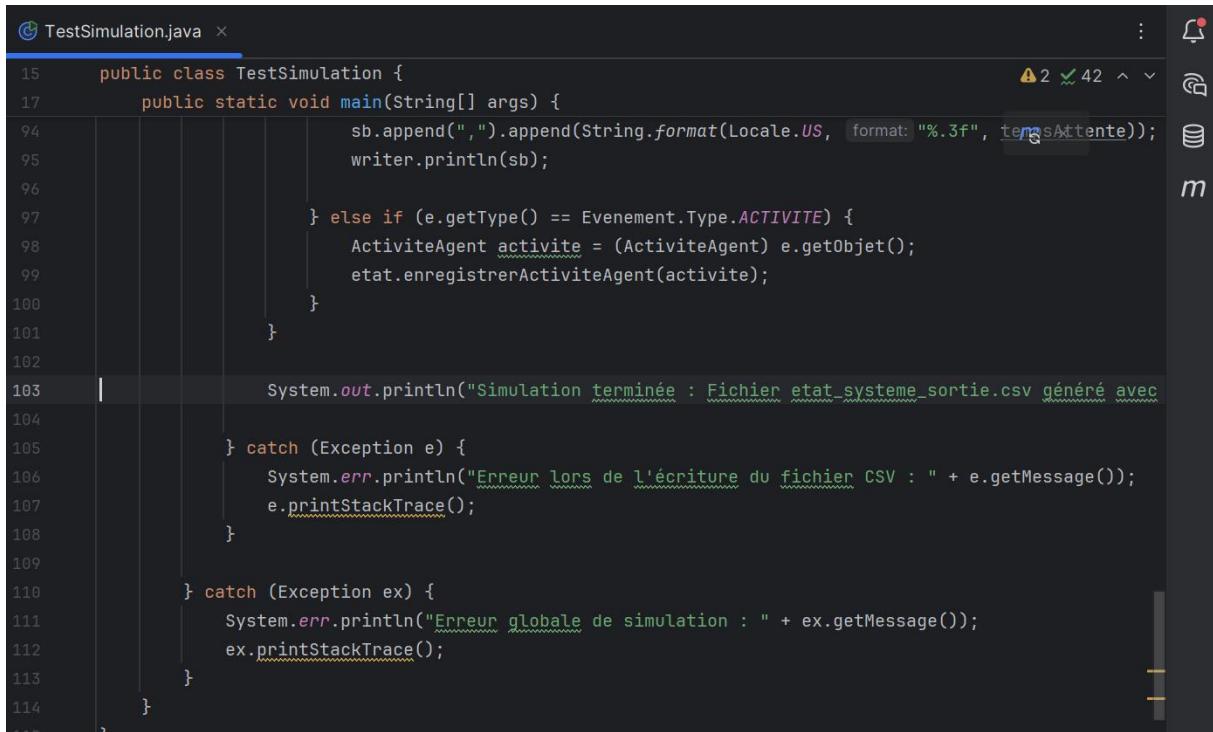
➤ But de la classe :

Représenter **un événement unique** dans le déroulement de la simulation (appel ou activité), avec :

- ◆ son **type** (APPEL ou ACTIVITE),
- ◆ sa **date exacte** (objet LocalDateTime),
- ◆ son **contenu associé** (Appel ou ActiviteAgent).

Ces objets sont ensuite stockés dans une **file de priorité** (PriorityQueue<Evenement>) triée **par date croissante**.

La classe **TestSimulation** orchestre l'exécution complète d'une simulation temporelle fidèle à partir des données historiques du centre d'appel VANAD. Elle permet de rejouer les événements (appels et activités) dans l'ordre chronologique et de capturer l'évolution de l'état du système.



```

15  public class TestSimulation {
16      public static void main(String[] args) {
17          StringWriter writer = new StringWriter();
18          Locale.US.setDisplayNames(true);
19          try {
20              for (Evenement e : evenements) {
21                  if (e.getType() == Evenement.Type.CALL) {
22                      String tempsAttente = String.format(Locale.US, "Temps d'attente : %.3f", tempsAttente);
23                      writer.append(",").append(tempsAttente);
24                      tempsAttente += 1;
25                  } else if (e.getType() == Evenement.Type.ACTIVITE) {
26                      ActiviteAgent activite = (ActiviteAgent) e.getObjet();
27                      etat.enregistrerActiviteAgent(activite);
28                  }
29              }
30              System.out.println("Simulation terminée : Fichier etat_systeme_sortie.csv généré avec succès");
31          } catch (Exception e) {
32              System.err.println("Erreur lors de l'écriture du fichier CSV : " + e.getMessage());
33              e.printStackTrace();
34          }
35      } catch (Exception ex) {
36          System.err.println("Erreur globale de simulation : " + ex.getMessage());
37          ex.printStackTrace();
38      }
39  }

```

➤ Rôles principaux :

- ◆ **Reconstituer fidèlement le déroulement historique** du centre d'appel à partir de deux flux d'événements : appels reçus et activités d'agents.
- ◆ **Capturer l'état du système** à chaque événement d'appel sous forme de vecteur de caractéristiques pour l'apprentissage automatique.
- ◆ **Calculer les prédicteurs heuristiques** du temps d'attente (LES et moyenne LES).
- ◆ **Générer un fichier CSV d'apprentissage** contenant les caractéristiques de chaque appel et le temps d'attente réellement observé.

➤ Fonctionnalités principales :

- ◆ **Lecture et tri des événements** : charge les appels et activités depuis des fichiers CSV, et les ordonne selon leur date d'occurrence.
- ◆ **Simulation pas à pas** : traite chaque événement, met à jour l'état du système (agents libres, files, horodatage), et enregistre les données utiles.
- ◆ **Calcul en ligne des prédicteurs** :

- ✧ *LES* (*Linear Exponential Smoothing*) : lissage exponentiel du temps d'attente.
- ✧ *Avg-LES* : moyenne des valeurs LES.
- ◆ **Export CSV** : génère un fichier `etat_systeme_sortie.csv` contenant les vecteurs caractéristiques accompagnés des temps d'attente.

➤ **Attributs clés :**

- ◆ `etat` : objet représentant l'état courant du système (agents disponibles, files, horodatage, etc.).
- ◆ `evenements` : liste triée chronologiquement d'objets Evenement (soit un appel, soit une activité).
- ◆ `alpha` : paramètre de lissage utilisé pour le calcul du prédicteur LES.
- ◆ `lesPrecedent`, `sommeLES`, `compteurLES` : variables intermédiaires pour calculer LES et sa moyenne.

➤ **Utilisation dans le projet :**

- ◆ Cette classe est utilisée pour **simuler de manière déterministe** le comportement du centre d'appel à partir de données réelles.
- ◆ Elle génère un **jeu d'apprentissage prêt à l'emploi** pour entraîner un modèle de prédiction du temps d'attente (ex. : ANN).
- ◆ Elle permet aussi de **valider les prédicteurs heuristiques** en les comparant au temps d'attente réellement observé (via le fichier CSV produit).

III. Prévisualisation et exécution de la simulation

Après avoir préparé les données nettoyées dans le notebook Python, nous avons implémenté et exécuté le projet **SimulationCentreAppelsVANAD** en Java, qui repose sur un moteur de **replay temporel fidèle**. Ce moteur permet de rejouer les événements (appels et activités) dans l'ordre chronologique et de générer un **jeu de données final** prêt pour l'entraînement d'un modèle prédictif.

Si vous souhaitez avoir une **vue plus détaillée** du projet, vous pouvez l'ouvrir dans l'environnement Java (ex. IntelliJ, VS Code, Eclipse). Le dossier `data_csv` inclus dans ce projet contient les **fichiers nettoyés des appels et activités**, obtenus via le notebook Python.

Le fichier de sortie `etat_systeme_sortie.csv` est automatiquement généré à la fin de l'exécution. Il contient pour chaque appel :

- ◆ les caractéristiques du système au moment de l'arrivée de l'appel,
- ◆ les prédictions issues des estimateurs LES et Avg-LES,
- ◆ le temps d'attente observé.

Lors de l'exécution du programme, les événements sont affichés progressivement dans la console.

SC SimulationCentreAppelsVANAD

Project SimulationCentreAppelsVANAD [centre-app]

TestSimulation.java

```

1 package Simulation;
2
3 import Donnees.LecteurCSV;
4 import Modele.Appel;

```

Run TestSimulation

C:\Users\bellj.jdk\ms-17.0.15\bin\java.exe ...

[APPEL] Nombre total d'appels lus : 1442772

[ACTIVITÉ] Nombre total d'activités lues : 1509997

Appels lus : 1442772

Activités lues : 1509997

Performance

Start Recording

00:00

CPU

Heap Memory

70:15 CRLF UTF-8 4 spaces

31°C Eclaircies 16:12

SC SimulationCentreAppelsVANAD

Version control

Project SimulationCentreAppelsVANAD [centre-app]

TestSimulation.java

etat_systeme_sortie.csv

SimulationCentreAppels.java

Run TestSimulation

Agent 9508 signalé libre à 2014-02-27T11:19:41

Appel reçu à 2014-02-27T11:19:42 pris en charge par l'agent 8594

Agent 8730 signalé libre à 2014-02-27T11:19:43

Agent 8594 signalé libre à 2014-02-27T11:19:43

Appel reçu à 2014-02-27T11:19:44 pris en charge par l'agent 8594

Agent 9576 signalé libre à 2014-02-27T11:19:48

Appel reçu à 2014-02-27T11:19:49 pris en charge par l'agent 8730

Agent 9576 signalé libre à 2014-02-27T11:19:50

Agent 8514 signalé libre à 2014-02-27T11:19:52

Appel reçu à 2014-02-27T11:19:53 pris en charge par l'agent 8514

Agent 7011 signalé libre à 2014-02-27T11:19:54

Appel reçu à 2014-02-27T11:19:56 pris en charge par l'agent 7011

Agent 7011 signalé libre à 2014-02-27T11:19:58

Agent 8730 signalé libre à 2014-02-27T11:19:58

Agent 8384 signalé libre à 2014-02-27T11:20:02

Agent 8384 signalé libre à 2014-02-27T11:20:02

Appel reçu à 2014-02-27T11:20:07 pris en charge par l'agent 8384

Appel reçu à 2014-02-27T11:20:07 pris en charge par l'agent 8730

Agent 8374 signalé libre à 2014-02-27T11:20:14

Appel reçu à 2014-02-27T11:20:18 pris en charge par l'agent 7011

Appel reçu à 2014-02-27T11:20:22 pris en charge par l'agent 9508

Appel reçu à 2014-02-27T11:20:24 pris en charge par l'agent 9576

Performance

Start Recording

00:00

CPU

Heap Memory

95:44 CRLF UTF-8 4 spaces

28°C Temps dégagé 00:28

```

package Simulation;
import Donnees.LecteurCSV;
import Modele.Appel;
import Modele.ActiviteAgent;
import java.io.FileWriter;

```

Appel en file traité par l'agent 7039 (appel initial à 2014-11-14T16:33:02)
 Appel reçu à 2014-11-14T17:22:06 mis en attente (file d'attente)
 Appel reçu à 2014-11-14T17:22:53 mis en attente (file d'attente)
 Appel reçu à 2014-11-14T17:23:42 mis en attente (file d'attente)
 Agent 6945 signalé libre à 2014-11-14T17:23:53
 Appel en file traité par l'agent 6945 (appel initial à 2014-11-14T16:33:10)
 Appel reçu à 2014-11-14T17:24:22 mis en attente (file d'attente)
 Appel reçu à 2014-11-14T17:24:47 mis en attente (file d'attente)
 Agent 6916 signalé libre à 2014-11-14T17:25:17
 Appel en file traité par l'agent 6916 (appel initial à 2014-11-14T16:33:18)
 Appel reçu à 2014-11-14T17:26:17 mis en attente (file d'attente)
 Appel reçu à 2014-11-14T17:26:32 mis en attente (file d'attente)
 Appel reçu à 2014-11-14T17:26:36 mis en attente (file d'attente)
 Appel reçu à 2014-11-14T17:26:51 mis en attente (file d'attente)
 Agent 6945 signalé libre à 2014-11-14T17:26:58

Une fois la simulation terminée, un message de confirmation est affiché :

- ◆ **Simulation terminée : etat_systeme_sortie.csv généré avec succès**

```

Appel en file traité par l'agent 6945 (appel initial à 2014-12-31T16:46:26)
Appel reçu à 2014-12-31T19:53:26 mis en attente (file d'attente)
Appel reçu à 2014-12-31T19:53:33 mis en attente (file d'attente)
Agent 9828 signalé libre à 2014-12-31T19:55:17
Appel en file traité par l'agent 9828 (appel initial à 2014-12-31T16:46:46)
Agent 9828 signalé libre à 2014-12-31T19:55:18
Appel en file traité par l'agent 9828 (appel initial à 2014-12-31T16:47:17)
Agent 1105 signalé libre à 2014-12-31T19:55:37
Appel en file traité par l'agent 1105 (appel initial à 2014-12-31T16:47:34)
Agent 1105 signalé libre à 2014-12-31T19:55:41
Appel en file traité par l'agent 1105 (appel initial à 2014-12-31T16:49:47)
Agent 6945 signalé libre à 2014-12-31T19:56:32
Appel en file traité par l'agent 6945 (appel initial à 2014-12-31T16:50:32)
Agent 1105 signalé libre à 2014-12-31T19:56:36
Appel en file traité par l'agent 1105 (appel initial à 2014-12-31T16:52:06)
Agent 9828 signalé libre à 2014-12-31T19:58:03
Appel en file traité par l'agent 9828 (appel initial à 2014-12-31T16:52:10)
Agent 9828 signalé libre à 2014-12-31T19:58:04
Appel en file traité par l'agent 9828 (appel initial à 2014-12-31T16:52:19)
Simulation terminée : Fichier etat_systeme_sortie.csv généré avec succès.

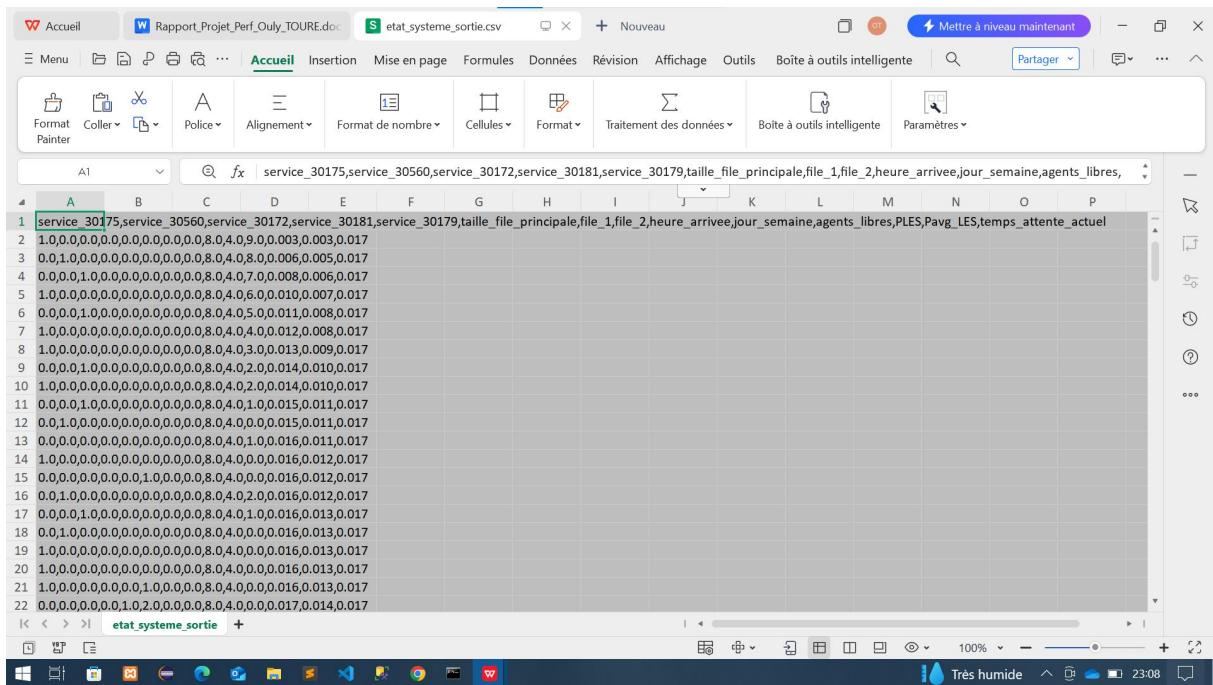
Process finished with exit code 0

```

L'exécution complète de la simulation a duré environ **2 minutes** sur ma machine de **8 Go de RAM**. Sur une machine plus puissante, le traitement peut être plus rapide.

Le fichier **etat_systeme_sortie.csv** est généré par le code Java de simulation du centre d'appels. Il contient les données d'état du système à différents instants, utilisées pour l'entraînement d'un modèle prédictif.

- ◆ Ce fichier est au format **CSV** (Comma-Separated Values), ce qui signifie que les valeurs sont séparées par des **virgules**.
 - ◆ Lorsqu'on ouvre ce fichier directement dans Excel par double-clic, les données apparaissent toutes dans une seule colonne, séparées par des virgules, ce qui rend la lecture difficile et confuse.



➤ Amélioration de la lisibilité des données avec Excel

Pour faciliter la lecture et l'analyse des données, j'ai procédé de la manière suivante :

- ◆ Au lieu d'ouvrir directement le fichier par double-clic, j'ai utilisé la fonctionnalité d'importation de données dans Excel.
 - ◆ J'ai importé le fichier via l'option **Données > À partir d'un fichier texte/CSV**.
 - ◆ Lors de l'importation, j'ai spécifié que le séparateur des données est la **virgule**.

- ◆ Excel a alors correctement réparti chaque valeur dans sa colonne propre, rendant le tableau clair et facilement exploitable.
- ◆ Cette méthode évite que les virgules soient affichées dans une même cellule et améliore considérablement la lisibilité du fichier.

The screenshot shows a Microsoft Excel spreadsheet titled "Classeur1 - Excel". The "Données" tab is selected. The data starts at row 1 with columns A through O. Row 1 contains headers such as "service_30175", "service_30560", "service_30172", "service_30181", "service_30179", "taille_file_principale", "file_1", "file_2", "heure_arrivee", "jour_semaine", "agents_libres", "PLES", "Pavg_LES", and "temps_attente_actuel". Rows 2 through 22 provide data points for these variables. The Windows taskbar at the bottom shows various icons and the date/time as 27°C Ciel couvert 23:00.

IV. Analyse exploratoire des données générées

Après exécution de la simulation Java, nous avons obtenu un fichier etat_systeme_sortie.csv contenant environ plus 1,4 million de lignes. Ce fichier décrit, à chaque arrivée d'appel, l'état du système ainsi que le délai d'attente observé.

Afin de préparer les données pour l'apprentissage automatique, une étape d'analyse exploratoire a été réalisée dans un notebook Python nommée Entrainement_Modele_ANN_Ouly_TOURE.ipynb. Les objectifs étaient de :

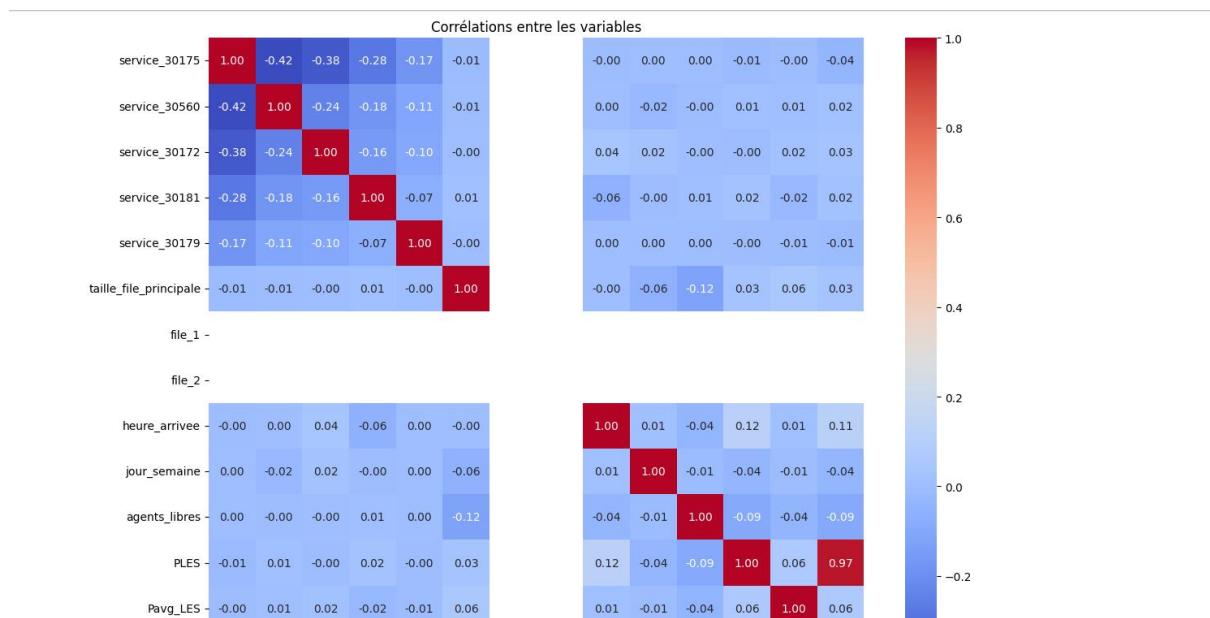
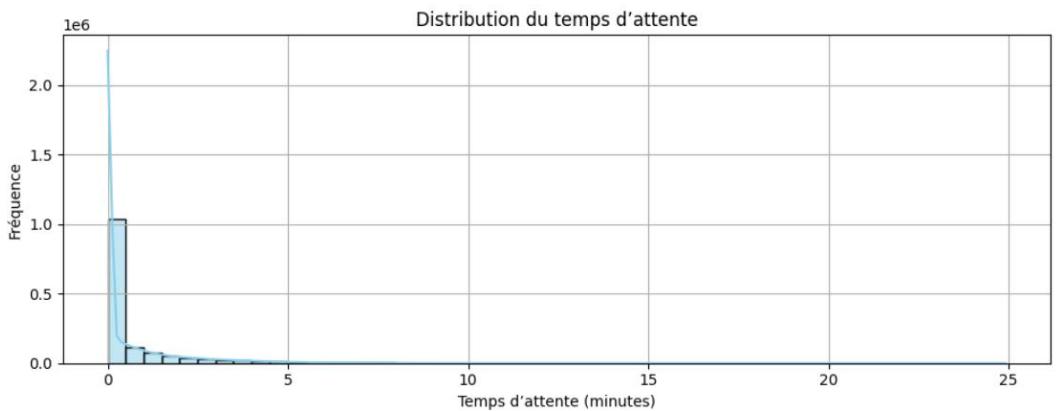
- ◆ vérifier la structure et l'intégrité des données,
- ◆ analyser la distribution du temps d'attente,

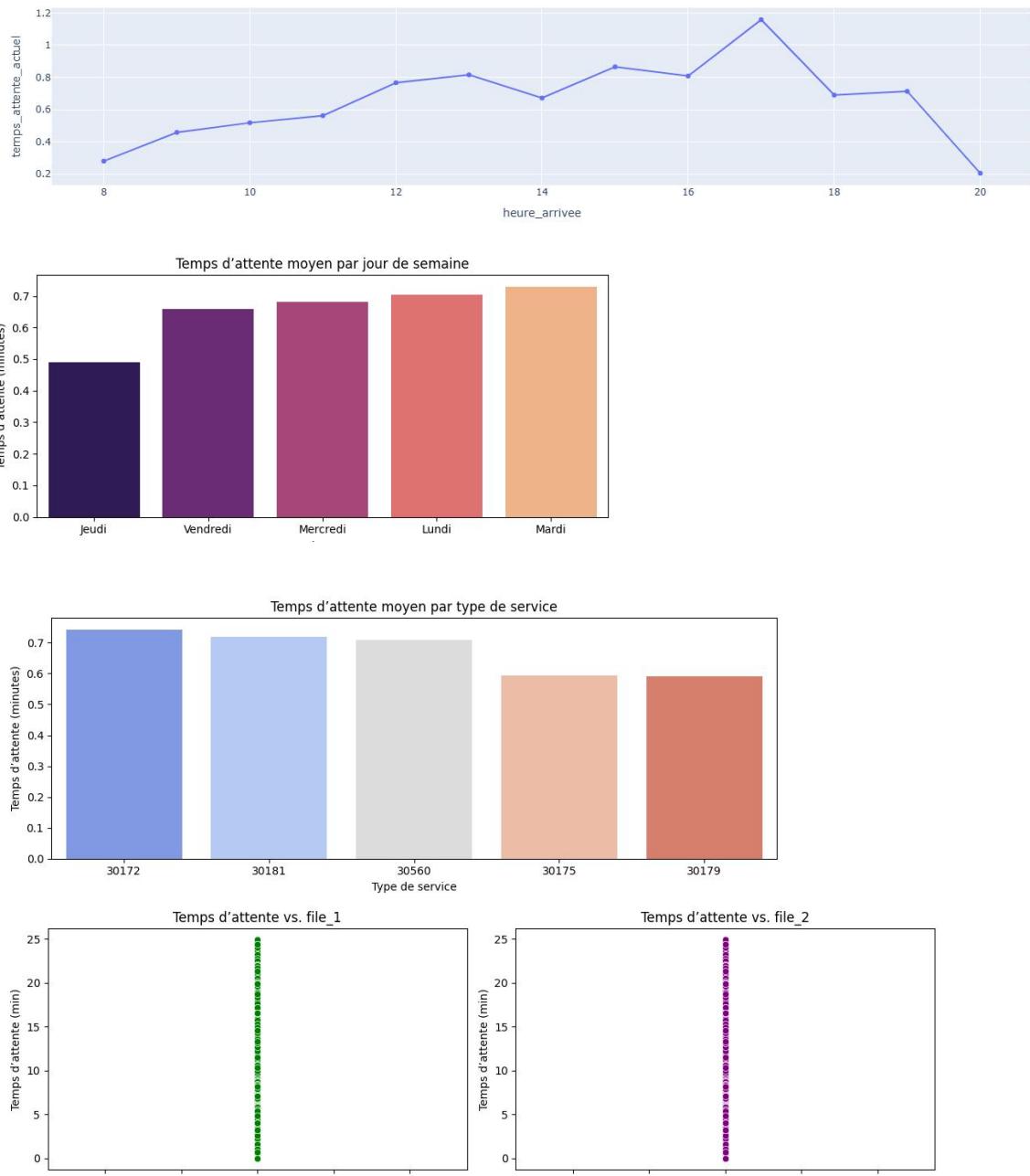
- ♦ observer les relations entre les prédicteurs (ex : heure, jour, agents libres) et la variable cible (`temps_attente_actuel`).

La figure ci-dessous illustre la distribution du délai d'attente, ainsi que sa variation en fonction de l'heure d'arrivée et du jour de la semaine. Une matrice de corrélation a également été calculée pour détecter les relations linéaires entre les variables.

Ces visualisations confirment la présence de tendances horaires et hebdomadaires, ainsi que l'impact du nombre d'agents disponibles sur le délai d'attente.

Aucun prédicteur ne présente de valeurs manquantes, ce qui valide la qualité du fichier généré par la simulation Java.





V. Modélisation par réseau de neurones (ANN)

- Importation des bibliothèques nécessaires

```
|: # Importation des bibliothèques nécessaires
import os
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from tensorflow import keras
from tensorflow.keras import layers
```

↑ ↓ ← → ⌂ ⌂ ⌂

Dans cette cellule, nous importons toutes les bibliothèques indispensables à la préparation des données, à la création du modèle de réseau de neurones (ANN), à l'entraînement, à l'évaluation et à la visualisation des résultats.

- ◆ os : gestion des chemins de fichiers.
- ◆ pandas, numpy : manipulation et traitement de données tabulaires et numériques.
- ◆ train_test_split (de sklearn) : pour diviser les données en ensembles d'entraînement et de test.
- ◆ StandardScaler : pour normaliser les données (centrées réduites).
- ◆ matplotlib.pyplot : pour les visualisations graphiques.
- ◆ mean_squared_error : pour évaluer les performances du modèle (MSE).
- ◆ keras / tensorflow.keras : pour définir, compiler, entraîner et évaluer le modèle de réseau de neurones.

➤ Chargement du fichier de données généré par la simulation Java

```
3]: # Chargement des données
chemin = r"C:\Users\DELL\Desktop\M2_BI_2025\Performance_des_SD\Projet_Perf_Ouly_TOURE_M2_BI_2025\SimulationCentreAppelsVANAD\etat_systeme_sortie.csv"
if not os.path.exists(chemin):
    raise FileNotFoundError(f"Fichier non trouvé : {chemin}")

df = pd.read_csv(chemin)

# Aperçu du jeu de données
print(df.head())
print(df.info())
print(df.isna().sum())
```

↑ ↓ ← → ⌂ ⌂ ⌂

Dans cette cellule, nous chargeons le fichier etat_systeme_sortie.csv, généré à l'issue de la simulation Java du centre d'appel VANAD. Ce fichier contient les données d'entrée pour l'entraînement du modèle ANN.

- ◆ chemin : chemin absolu vers le fichier CSV.

- ◆ pd.read_csv(...) : lecture du fichier avec pandas sous forme de DataFrame.
- ◆ df.head() : aperçu des premières lignes du jeu de données.
- ◆ df.info() : informations générales sur les colonnes, types et taille.
- ◆ df.isna().sum() : vérifie la présence éventuelle de valeurs manquantes.

But : s'assurer que les données ont bien été chargées et qu'elles sont complètes et correctement structurées avant de les utiliser pour la modélisation.

➤ Préparation des données pour l'entraînement

```
# Séparation X / y
X = df.drop(columns=["temps_attente_actuel"])
y = df["temps_attente_actuel"]

# Séparation en train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalisation
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Dans cette cellule, nous préparons les données pour entraîner notre réseau de neurones :

- ◆ **Séparation des variables d'entrée et de sortie :**
 - ✧ X contient toutes les colonnes **sauf** temps_attente_actuel → ce sont les **données d'entrée** (état du système).
 - ✧ y contient uniquement temps_attente_actuel → c'est la **valeur à prédire**.
- ◆ **Découpage en deux ensembles :**
 - ✧ train_test_split(...) divise les données en :
 - 80 %** pour l'entraînement (X_train, y_train)
 - 20 %** pour le test (X_test, y_test)

- ✧ random_state=42 permet de reproduire la même séparation à chaque exécution.

◆ **Normalisation des données :**

- ✧ On **standardise** les données (moyenne = 0, écart-type = 1) avec StandardScaler pour faciliter l'apprentissage du modèle.
- ✧ On applique cette transformation uniquement sur les **données d'entrée X**.

➤ **Création du modèle de réseau de neurones (ANN)**

```
[1]: # Création du modèle ANN
model = keras.Sequential([
    keras.Input(shape=(X_train_scaled.shape[1],)),
    layers.Dense(64, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(1)
])

model.compile(optimizer="adam", loss="mse")
model.summary()
```

↑ ↓ ← → ⌂

Dans cette cellule, nous définissons l'architecture du modèle ANN utilisé pour **prédir le temps d'attente** des appels :

◆ **Modèle Sequential :**

Le modèle est créé comme une **succession de couches**, de l'entrée à la sortie.

◆ **Structure du réseau :**

- ✧ keras.Input(...) : précise la **taille du vecteur d'entrée** (13 colonnes dans notre jeu de données).
- ✧ Dense(64, activation="relu") : première **couche cachée** avec 64 neurones et la fonction d'activation **ReLU** (Rectified Linear Unit).
- ✧ Dropout(0.3) : couche qui **désactive aléatoirement 30 % des neurones** à chaque itération pour **éviter le surapprentissage**.

- ✧ Dense(1) : couche de **sortie** avec un seul neurone (car on prédit une seule valeur continue : le temps d'attente).

◆ **Compilation du modèle :**

- ✧ optimizer="adam" : algorithme efficace pour optimiser les poids du réseau.
- ✧ loss="mse" : la fonction d'erreur utilisée est la **Mean Squared Error (MSE)**, adaptée à la régression.

◆ **Résumé du modèle (model.summary()) :**

Affiche la structure du réseau, le nombre de paramètres à apprendre, et la forme de chaque couche.

```
Model: "sequential"
-----  

Layer (type)      Output Shape        Param #
dense (Dense)    (None, 64)           896
dropout (Dropout) (None, 64)           0
dense_1 (Dense)   (None, 1)            65
-----  

Total params: 961 (3.75 KB)
Trainable params: 961 (3.75 KB)
Non-trainable params: 0 (0.00 B)
```

➤ **Entraînement du modèle**

```
# Entrainement du modèle
early_stopping = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)
history = model.fit(
    X_train_scaled, y_train,
    validation_split=0.2,
    epochs=100,
    batch_size=256,
    callbacks=[early_stopping],
    verbose=1
)
```



Dans cette cellule, nous entraînons le réseau de neurones sur les données d'apprentissage.

- ◆ model.fit(...) : lance l'entraînement du modèle.
 - ✧ **80 %** des données sont utilisées pour l'entraînement, **20 %** pour la validation (validation_split=0.2).
 - ✧ L'entraînement se fait pendant **jusqu'à 100 itérations (epochs)**.

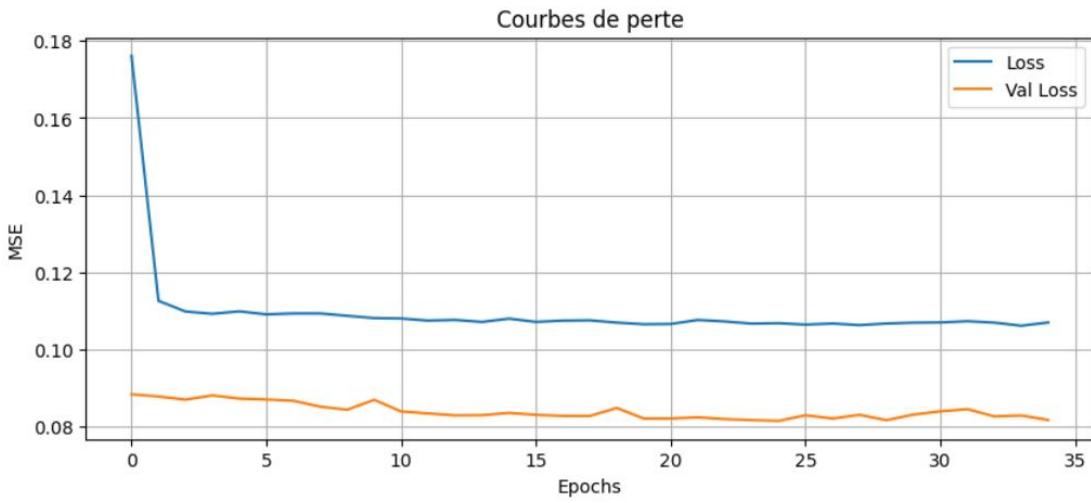
- ✧ **batch_size=256** : le modèle est mis à jour toutes les 256 lignes.
- ✧ **verbose=1** : affiche les détails de chaque epoch.
- ◆ **early_stopping** : arrête l'entraînement automatiquement si le modèle ne s'améliore plus pendant **10 epochs**, pour éviter le surapprentissage.
- ◆ Le résultat est stocké dans **history**, qui contient les courbes de perte pour analyse.

```
Epoch 1/100
3607/3607 ━━━━━━━━━━━━━━━━━━━━ 16s 4ms/step - loss: 0.3468 - val_loss: 0.0884
Epoch 2/100
3607/3607 ━━━━━━━━━━━━━━━━ 18s 5ms/step - loss: 0.1150 - val_loss: 0.0879
Epoch 3/100
3607/3607 ━━━━━━━━━━━━━━ 16s 4ms/step - loss: 0.1074 - val_loss: 0.0871
Epoch 4/100
3607/3607 ━━━━━━━━━━━━ 22s 5ms/step - loss: 0.1105 - val_loss: 0.0882
Epoch 5/100
3607/3607 ━━━━━━━━━━ 14s 4ms/step - loss: 0.1121 - val_loss: 0.0873
Epoch 6/100
3607/3607 ━━━━━━ 15s 4ms/step - loss: 0.1088 - val_loss: 0.0871
Epoch 7/100
3607/3607 ━━━━ 14s 4ms/step - loss: 0.1093 - val_loss: 0.0868
Epoch 8/100
3607/3607 ━━ 12s 3ms/step - loss: 0.1084 - val_loss: 0.0852
Epoch 9/100
-----
```

➤ L'évolution de l'erreur du modèle

```
]:# Courbes de perte
plt.figure(figsize=(10, 4))
plt.plot(history.history["loss"], label="Loss")
plt.plot(history.history["val_loss"], label="Val Loss")
plt.xlabel("Epochs")
plt.ylabel("MSE")
plt.legend()
plt.title("Courbes de perte")
plt.grid(True)
plt.show()
```

- ◆ Il trace deux courbes :
 - ✧ Loss : erreur sur les données d'entraînement.
 - ✧ Val Loss : erreur sur les données de validation.
- ◆ Cela permet de **voir si le modèle apprend bien** ou s'il commence à trop s'adapter (surapprentissage).
- ◆ L'axe X montre les **epochs** (les cycles d'entraînement).
- ◆ L'axe Y montre la **valeur de l'erreur MSE**.



➤ Évaluation des performances du modèle ANN

```
# Évaluation du modèle
y_pred = model.predict(X_test_scaled).flatten()

mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
rrmse = rmse / np.mean(y_test)

print(f'MSE: {mse:.3f}')
print(f'RMSE: {rmse:.3f} minutes')
print(f'RRMSE: {rrmse:.3f}')

# Prédictions vs valeurs réelles
plt.figure(figsize=(6, 6))
plt.scatter(y_test, y_pred, alpha=0.2)
plt.plot([0, max(y_test)], [0, max(y_test)], color="red", linestyle="--")
plt.xlabel("Vrai temps d'attente")
plt.ylabel("Temps prédict")
plt.title("Prédictions vs Valeurs réelles")
plt.grid(True)
plt.show()
```

◆ Prédiction :

Le modèle prédit les temps d'attente pour les données de test (X_test_scaled).

◆ Calcul des erreurs :

- ❖ MSE : l'erreur moyenne entre les vraies valeurs et les prédictions.
- ❖ RMSE : la racine carrée du MSE (plus facile à interpréter).
- ❖ RRMSE : le RMSE rapporté à la moyenne réelle, pour mieux comparer.

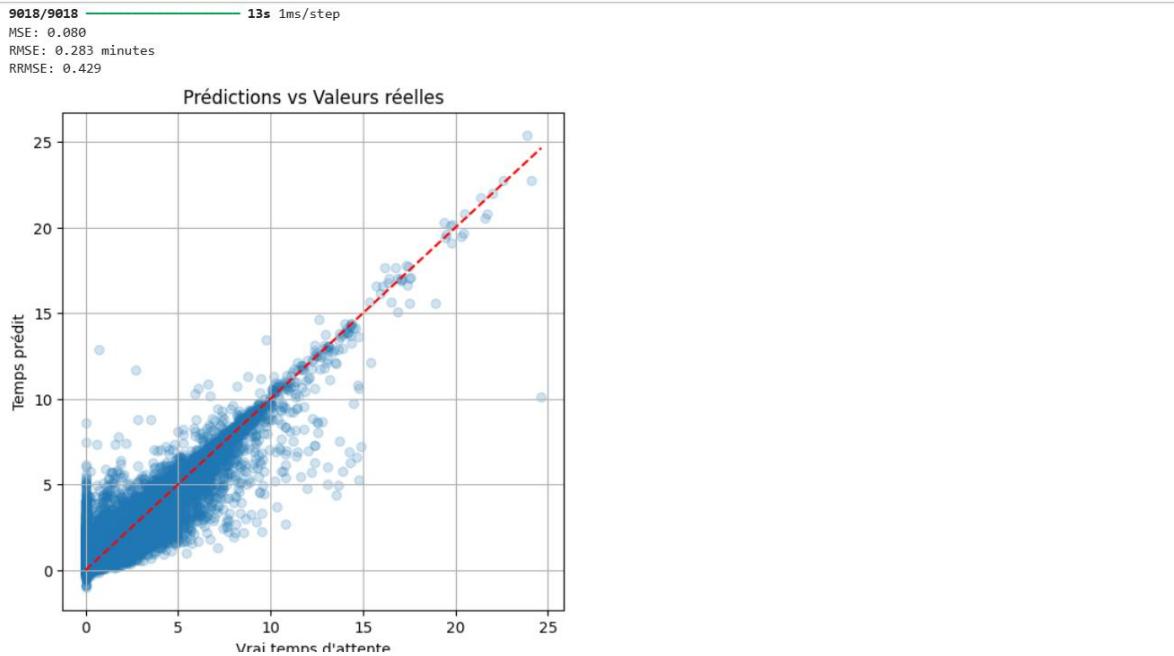
◆ Affichage des résultats :

Il affiche les erreurs calculées (MSE, RMSE, RRMSE) en minutes.

◆ Visualisation :

Le graphique montre la comparaison entre :

- ✧ les **valeurs réelles** (axe X),
- ✧ les **valeurs prédites** par le modèle (axe Y).
- ✧ La ligne rouge représente la perfection (si la prédiction était exacte).



➤ Affichage et Exportation du jeu de données:

```
[10]: # Affichage
print("\nRécapitulatif :")
print(f"Taille du jeu de test : {len(y_test)}")
print(f"Temps d'attente moyen (test) : {np.mean(y_test):.2f} min")
print(f"RMSE final : {rmse:.3f} min")

Récapitulatif :
Taille du jeu de test : 288555
Temps d'attente moyen (test) : 0.66 min
RMSE final : 0.283 min

[12]: # Export des prédictions
pred_df = pd.DataFrame({
    "attente_reelle": y_test,
    "prediction_ann": y_pred
})

chemin_export = r"C:\Users\DELL\Desktop\M2_BI_2025\Performance_des_SD\Projet_Perf_Ouly_TOURE_M2_BI_2025\predictions_ann.csv"
pred_df.to_csv(chemin_export, index=False)

print(f"Exportation réussie ! Fichier enregistré à : {chemin_export}")
Exportation réussie ! Fichier enregistré à : C:\Users\DELL\Desktop\M2_BI_2025\Performance_des_SD\Projet_Perf_Ouly_TOURE_M2_BI_2025\predictions_ann.csv
```

➤ Performance des Prédicteurs LES et Avg LES:

```
[1]: # Performance des prédicteurs LES et Avg-LES
import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error

# Charger le fichier complet généré par Java
df = pd.read_csv(r'C:\Users\DeLL\Desktop\M2_BI_2025\Performance_des_SD\Projet_Perf_Ouly_TOURE_M2_BI_2025\SimulationCentreAppelsVANAD\etat_systeme_sortie.csv')
# Vrai temps d'attente
y_true = df["temps_attente_actuel"]

# LES
mse_les = mean_squared_error(y_true, df["PLES"])
rmse_les = np.sqrt(mean_squared_error(y_true, df["PLES"]))
rrmse_les = rmse_les / np.mean(y_true)

# Avg-LES
mse_avg_les = mean_squared_error(y_true, df["Avg_LES"])
rmse_avg_les = np.sqrt(mean_squared_error(y_true, df["Avg_LES"]))
rrmse_avg_les = rmse_avg_les / np.mean(y_true)

print(f"LES - MSE : {mse_les:.3f} | RMSE : {rmse_les:.3f} min | RRMSE : {rrmse_les:.3f}")
print(f"Avg-LES - MSE : {mse_avg_les:.3f} | RMSE : {rmse_avg_les:.3f} min | RRMSE : {rrmse_avg_les:.3f}")

LES - MSE : 0.094 | RMSE : 0.306 min | RRMSE : 0.465
Avg-LES - MSE : 1.773 | RMSE : 1.332 min | RRMSE : 2.023
```

VI. Comparaison des modèles

Modèle	MSE	RMSE (minutes)	RRMSE	Observations
LES	0.094	0.306	0.465	Réagit vite mais sensible aux variations
Avg-LES	1.773	1.332	2.023	Plus stable mais moins réactif
ANN	0.080	0.283	0.429	Plus précis, surtout en cas de files longues

Le réseau de neurones (ANN) est le prédicteur le plus précis, avec une erreur plus faible que LES et Avg-LES. Il donne de meilleurs résultats surtout dans les situations complexes, lorsque plusieurs facteurs influencent le temps d'attente. Le prédicteur LES est rapide mais instable, car il repose uniquement sur la dernière valeur observée, ce qui le rend sensible aux variations brusques. De son côté, Avg-LES est plus stable, mais il réagit plus lentement aux changements. L'ANN, en utilisant plusieurs variables comme le type de service, l'heure, le nombre d'agents disponibles ou les files d'attente, apprend à mieux modéliser le comportement global du système. En combinant LES ou Avg-LES avec l'ANN, il est possible d'obtenir un modèle à la fois réactif et précis, adapté à différents contextes opérationnels.

Conclusion

Ce projet a permis d'implémenter et de comparer trois prédicteurs du temps d'attente dans un centre d'appel multi-compétences : LES, Avg-LES et un réseau de neurones artificiel (ANN). À partir des données réelles du centre VANAD, nous avons simulé le système pour reconstituer l'état à chaque arrivée d'appel, puis entraîné un modèle ANN pour prédire le délai d'attente.

Les résultats montrent que les prédicteurs LES et Avg-LES fournissent des approximations utiles, avec un compromis entre rapidité de réaction et stabilité. Cependant, l'ANN dépasse ces méthodes en précision grâce à sa capacité à exploiter de multiples variables et à modéliser des relations complexes.

Cette étude confirme le potentiel des techniques d'apprentissage machine pour améliorer la gestion des centres d'appels, en proposant des estimations plus fiables du délai d'attente client. Elle ouvre la voie à l'intégration de modèles prédictifs avancés dans les outils opérationnels, et à des développements futurs comme l'ajout de facteurs saisonniers, la création d'interfaces prédictives ou l'utilisation d'autres algorithmes d'intelligence artificielle.