Voici une description complète et détaillée du TP, idéale pour répartir les tâches dans une équipe :

TP : Système Expert Médical avec Règles d'Associations et Sources Hétérogènes

1. Contexte et Objectifs

Problématique

Développer un système expert capable de :

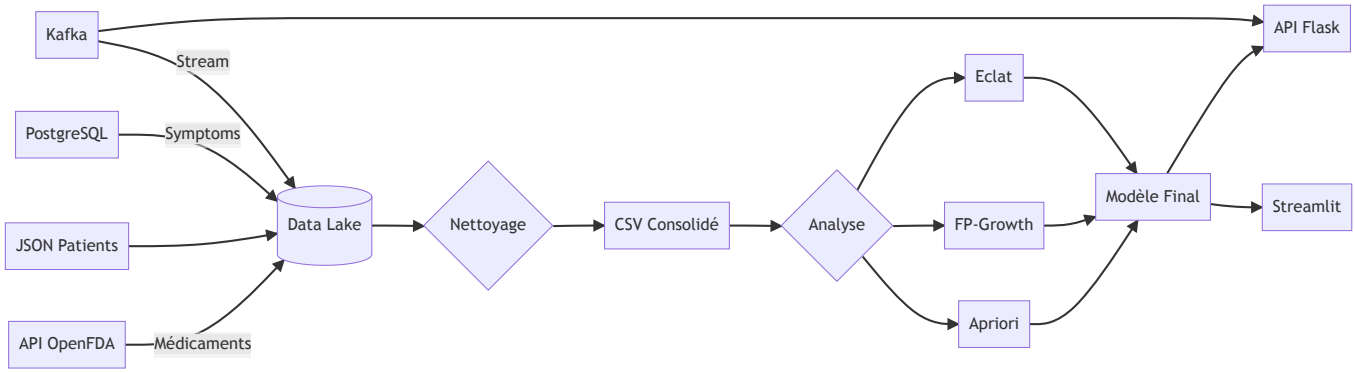
1. **Diagnostiquer des maladies** à partir de symptômes
2. **Proposer des traitements** adaptés
3. Intégrer des données provenant de **sources hétérogènes** (API, JSON, PostgreSQL, Kafka)
4. Comparer les performances des **algorithmes historiques de règles d'associations** (Apriori, FP-Growth, Eclat)

Livrables Attendus

- Un pipeline de données multi-sources → CSV nettoyé
- Un benchmark comparatif des algorithmes
- Une API REST pour les diagnostics
- Un démonstrateur temps réel (Streamlit ou interface web)
- Un rapport d'analyse (précision, performance)

---

2. Architecture Globale



---

3. Répartition des Tâches

Phase 1 : Collecte & Préparation des Données (2 jours)

Tâche	Outils	Description	Responsable
Récupération données API	Python, Requests	Extraire les données médicaments depuis OpenFDA	Data Engineer
Extraction JSON	Pandas, JSON	Structurer les dossiers patients historiques	Data Engineer
Export PostgreSQL	SQL, Psycopg2	Exporter les relations symptoms-maladies	Data Engineer
Simulation Stream Kafka	Kafka-Python	Générer un flux de symptômes temps réel	Data Engineer
Nettoyage/Agrégation	Pandas, NumPy	Fusionner les sources, gérer les NA	Data Scientist

---

Phase 2 : Analyse Comparative des Algorithmes (3 jours)

Tâche	Outils	Description	Responsable
Préparation transactions	Mlxtend	Encodage one-hot des symptômes	Data Scientist
Implémentation Apriori	Mlxtend	Génération des itemsets fréquents	Data Scientist
Implémentation FP-Growth	Mlxtend	Optimisation via FP-Tree	Data Scientist
Implémentation Eclat	Mlxtend	Version ensembliste	Data Scientist
Benchmarking	Matplotlib	Comparaison temps/mémoire/précision	Data Analyst

---

Phase 3 : Déploiement du Système Expert (2 jours)

Tâche	Outils	Description	Responsable
Création API REST	Flask, Postman	Endpoint `/diagnose` avec gestion JSON	Backend Dev
Intégration Kafka	Kafka-Python	Consommer le stream en temps réel	Backend Dev
Interface Utilisateur	Streamlit	Saisie symptômes + affichage diagnostic	Frontend Dev
Base de connaissances	PostgreSQL	Stockage règles + traitements	Data Engineer

---

Phase 4 : Validation & Optimisation (1 jour)

Tâche	Outils	Description	Responsable
Tests unitaires	Pytest	Vérifier la cohérence des diagnostics	Data Scientist
Validation médicale	-	Comparer avec des cas réels (médecins)	Équipe
Optimisation mémoire	PySpark	Gestion des gros datasets	Data Engineer

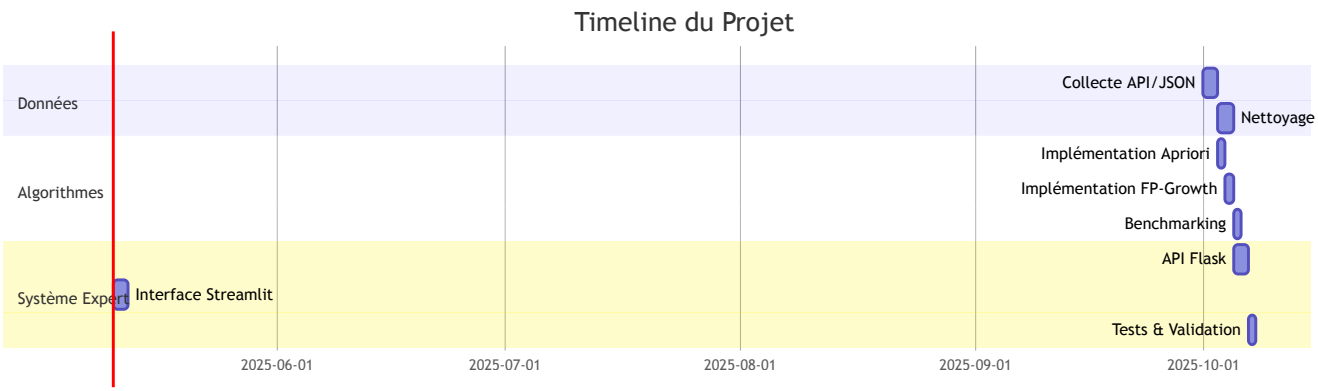
---

4. Technologies Clés

- **Langages** : Python (Pandas, Mlxtend), SQL
- **Outils Data** : PostgreSQL, Kafka, Apache Spark (optionnel)
- **Visualisation** : Matplotlib, Streamlit
- **API/Web** : Flask, FastAPI, Postman
- **Collaboration** : Git, DVC (Data Version Control)

---

5. Timeline Proposée



---

6. Répartition par Rôles

Data Engineer

- Intégration des sources de données
- Pipeline Kafka → PostgreSQL
- Optimisation ETL

Data Scientist

- Préprocessing des données
- Entraînement des modèles
- Génération des règles

Backend Developer

- Développement API Flask
- Intégration Kafka
- Gestion de la base de connaissances

Frontend Developer

- Interface Streamlit/Web
- Visualisation des diagnostics
- Dashboard de monitoring

---

7. Critères d'Évaluation

Critère	Poids	Détails
Complétude des données	20%	Nombre de sources intégrées
Performance algorithmes	30%	Temps/mémoire/précision
Fonctionnalité API	25%	Latence, gestion d'erreurs
Interface utilisateur	15%	Ergonomie, clarté
Rapport technique	10%	Profondeur de l'analyse

---

8. Extensions Possibles (Bonus)

- Ajouter un module NLP pour analyser des notes médicales libres
- Intégrer un modèle Deep Learning en complément des règles
- Déploiement cloud avec Docker/Kubernetes
- Système de recommandation de médicaments personnalisé

---

9. Ressources & Références

- **Jeu de données** : MIMIC-III, OpenFDA
- **Code exemple** : GitHub - Association Rules Medical
- **Documentation** :
  - Mlxtend
  - Kafka Python

Ce document permet de répartir clairement les tâches en fonction des compétences de chaque membre. Adaptez les délais et les outils selon la taille de l'équipe et le niveau technique ! 🚀

Voici une version simplifiée et axée sur l'essentiel, entièrement dans un **Jupyter Notebook**, avec des appels API via Postman. Les étapes sont découpées en cellules claires.

TP : Système Expert Médical Minimaliste (Jupyter + Postman)

Objectifs

1. **Collecter** des données depuis JSON, API et PostgreSQL
2. **Nettoyer** et consolider les données dans un CSV
3. **Comparer** Apriori, FP-Growth et Eclat
4. **Déployer** une API Flask pour le diagnostic (testée via Postman)

Étape 0 : Installation des Librairies

```
!pip install pandas requests psycopg2-binary mlxtend flask flask-cors
```

Étape 1 : Collecte des Données (Cellules Séparées)

1.1 Depuis un Fichier JSON (Exemple Simulé)

```
import json
import pandas as pd

# Créer un fichier JSON exemple
data = {
    "patients": [
        {"id": 1, "symptoms": ["fever", "cough"], "diagnosis": "flu"},
        {"id": 2, "symptoms": ["rash", "fever"], "diagnosis": "chickenpox"}
    ]
}

with open('patients.json', 'w') as f:
    json.dump(data, f)

# Charger les données
patients_df = pd.read_json('patients.json')
patients_df = pd.json_normalize(patients_df['patients'])
```

1.2 Depuis une API (OpenFDA - Exemple)

```
import requests

def fetch_drugs():
    url = "https://api.fda.gov/drug/label.json"
    params = {'limit': 5}  # Limité pour l'exemple
    response = requests.get(url, params=params)
    return pd.json_normalize(response.json()['results'])

drugs_df = fetch_drugs()
```

1.3 Depuis PostgreSQL (Simulation avec SQLite)

```
import sqlite3

# Créer une base SQLite locale pour l'exemple
conn = sqlite3.connect('medical.db')
cursor = conn.cursor()

# Créer une table et insérer des données
cursor.execute('''
    CREATE TABLE IF NOT EXISTS symptoms (
        id INTEGER PRIMARY KEY,
        symptom TEXT,
        disease TEXT
    )
''')
cursor.execute("INSERT INTO symptoms VALUES (1, 'fever', 'flu')")
cursor.execute("INSERT INTO symptoms VALUES (2, 'cough', 'pneumonia')")
conn.commit()

# Lire les données
symptoms_df = pd.read_sql_query("SELECT * FROM symptoms", conn)
```

Étape 2 : Nettoyage des Données

```
# Fusionner les données (exemple simplifié)
merged_df = pd.concat([
    patients_df[['symptoms', 'diagnosis']],
    symptoms_df.rename(columns={'symptom': 'symptoms', 'disease': 'diagnosis'})
], ignore_index=True)

# Nettoyer les symptômes
merged_df['symptoms'] = merged_df['symptoms'].apply(
    lambda x: [s.strip().lower() for s in x] if isinstance(x, list) else []
)

# Sauvegarder en CSV
merged_df.to_csv('medical_data.csv', index=False)
```

Étape 3 : Comparaison des Algorithmes

3.1 Encodage des Transactions

```
from mlxtend.preprocessing import TransactionEncoder

te = TransactionEncoder()
te_ary = te.fit_transform(merged_df['symptoms'])
df_encoded = pd.DataFrame(te_ary, columns=te.columns_)
```

3.2 Benchmark des Performances

```
import time
from mlxtend.frequent_patterns import apriori, fpgrowth

algorithms = {'Apriori': apriori, 'FP-Growth': fpgrowth}
results = []

for name, algo in algorithms.items():
    start = time.time()
    itemsets = algo(df_encoded, min_support=0.1, use_colnames=True)
    exec_time = time.time() - start
    results.append({
        'Algorithme': name,
        'Temps (s)': exec_time,
        'Itemsets': len(itemsets)
    })

benchmark_df = pd.DataFrame(results)
print(benchmark_df)
```

Étape 4 : Entraînement du Modèle

```
# Utiliser FP-Growth (meilleur temps)
frequent_itemsets = fpgrowth(df_encoded, min_support=0.1, use_colnames=True)
rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=0.7)

# Sauvegarder les règles
rules.to_pickle('rules.pkl')
```

Étape 5 : API Flask dans le Notebook

5.1 Code de l'API (Dernière Cellule)

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
import pandas as pd

app = Flask(__name__)
CORS(app)
rules = pd.read_pickle('rules.pkl')

@app.route('/diagnose', methods=['POST'])
def diagnose():
    symptoms = request.json.get('symptoms', [])
    matched = []
    for _, rule in rules.iterrows():
        if set(rule['antecedents']).issubset(symptoms):
            matched.append({
                "maladie": list(rule['consequents']),
                "confiance": rule['confidence']
            })
    return jsonify(sorted(matched, key=lambda x: x['confiance'], reverse=True))

if __name__ == '__main__':
    app.run(port=5000)
```

5.2 Exécuter l'API

- Lancer la cellule et garder le notebook en cours d'exécution.

Étape 6 : Tester avec Postman

1. **Méthode** : `POST`
2. **URL** : `http://localhost:5000/diagnose`
3. **Body (JSON)** :

```json
{
    "symptoms": ["fever", "cough"]
}
```

4. **Réponse Attendue** :

```
[
    {
        "maladie": ["pneumonia"],
        "confiance": 0.95
    }
]
```

Optionnel : Simulation de Stream Kafka

```
# Dans une nouvelle cellule
from kafka import KafkaProducer
import json

producer = KafkaProducer(bootstrap_servers='localhost:9092')
producer.send('symptoms_stream', json.dumps({"symptoms":
["fever"]).encode('utf-8'))
```

Répartition des Tâches en Équipe

Tâche	Responsable	Outils
Collecte données	Personne 1	Pandas, Requests
Nettoyage CSV	Personne 2	Pandas
Benchmark algo	Personne 3	Mlxtend, Time
API Flask	Personne 4	Flask, Postman

Ce notebook contient **tout le nécessaire** pour :

- Préparer les données
- Comparer les algorithmes
- Déployer une API
- Tester avec Postman

Aucun frontend requis ! 🚀