Rapport d'Épreuve E1 : Construction du Pipeline de Données (C1-C5)

Candidat : Rida Boualam **Projet** : Bitcoin Analyzer **Date** : Juillet 2025

Introduction

Ce premier rapport explique comment j'ai construit la base du projet "**Bitcoin Analyzer**" : le pipeline de données. L'idée était simple : mettre en place un système automatique et fiable pour aller chercher les informations, les nettoyer, les stocker proprement, et enfin les rendre accessibles via une API. C'est le socle sur lequel toute l'intelligence artificielle du projet va reposer.

C1 : Collecter les Données à la Source

Pour que l'analyse soit pertinente, il me fallait deux types de données : les chiffres du marché et les actualités. J'ai donc mis en place deux collecteurs automatiques.

Source 1 : Les données de marché via l'API Coinalyze

Pour les données financières, la fiabilité est clé. C'est pourquoi j'ai choisi une API reconnue, **Coinalyze**. Dans le script `scripts/extraction_api.py`, j'utilise la bibliothèque `requests` pour interroger leur service. La sécurité était une priorité : la clé d'API est stockée dans un fichier `.env`, qui n'est pas envoyé sur GitHub, et mon code la charge sans jamais l'écrire en dur. Ainsi, la clé ne se retrouve jamais dans le code partagé.

Extrait de `scripts/extraction_api.py` : la méthode sécurisée pour appeler l'API.

```
# Fichier: scripts/extraction_api.py
from dotenv import load_dotenv
import os
import requests

# Je charge les variables du fichier .env
load_dotenv()
API_KEY = os.getenv("COINALYZE_API")

def get_bitcoin_data():
    # J'utilise la clé chargée en mémoire pour m'authentifier
    headers = {"api_key" : API_KEY}
    params = {
        "symbols": "BTCUSDC.A",
        "interval": "1hour"
    }
    response = requests.get(API_URL, headers=headers, params=params)

    # Je vérifie que tout s'est bien passé avant de continuer
    if response.status_code == 200:
        return response.json()
    else:
```

```
        # En cas d'erreur, j'affiche un message clair pour le debug
        print(f"Erreur {response.status_code} : {response.text}")
        return None
```

Source 2 : Les actualités par scraping

Pour les actualités, c'était plus complexe car il n'y avait pas d'API simple. Les sites modernes utilisent beaucoup de JavaScript pour charger leur contenu, ce qui bloque les scrapers classiques. La solution a donc été de simuler un vrai navigateur. J'ai utilisé `undetected-chromedriver` pour lancer une instance de Chrome, attendre que la page se charge complètement, et seulement après, j'ai analysé le HTML avec `BeautifulSoup` pour en extraire les titres et les liens.

Extrait de `scripts/extraction_news.py` : la technique pour scraper un site dynamique.

```python
# Fichier: scripts/extraction_news.py
import time
import undetected_chromedriver as uc
from bs4 import BeautifulSoup

def extract_news_with_browser():
    # Je lance un vrai navigateur en arrière-plan
    driver = uc.Chrome()
    driver.get("https://news.bitcoin.com/")

    # C'est l'étape clé : je laisse le temps au JavaScript de charger le contenu
    time.sleep(5)

    # Maintenant, je peux récupérer le HTML final et le parser
    html_content = driver.page_source
    soup = BeautifulSoup(html_content, 'html.parser')
    # ... (logique de parsing)
    driver.quit()
```

C2 : Interroger un Système Existant avec SQL

En entreprise, on doit souvent se connecter à des bases de données qui existent déjà. Pour simuler ce cas, j'ai créé une petite base de données "legacy" (`source_data.db`) avec quelques articles. Ensuite, mon script `scripts/extraction_sql.py` s'y connecte et utilise une requête `SELECT` toute simple pour récupérer les informations. C'est une compétence de base mais essentielle.

Extrait de `scripts/extraction_sql.py` : la requête SQL pour extraire les données.

```python
# Fichier: scripts/extraction_sql.py
import sqlite3

# Connexion à la base de données legacy
source_conn = sqlite3.connect("data/source_data.db")
source_cursor = source_conn.cursor()
```

```
# J'exécute une requête SQL standard pour récupérer ce qui m'intéresse
query = "SELECT article_title, article_url FROM legacy_articles;"
source_cursor.execute(query)
articles_from_source = source_cursor.fetchall()
source_conn.close()
```

C4 : Structurer et Stocker les Données de Manière Fiable

Une fois les données collectées, il fallait les stocker. J'ai choisi **SQLite** parce que c'est parfait pour démarrer un projet : c'est un simple fichier, ça ne demande pas d'installer un serveur lourd comme **PostgreSQL**, et c'est intégré à Python.

Dans mon script `scripts/stockage.py`, j'ai défini la structure des tables. Le point le plus important était de garantir l'intégrité des données. Pour cela, j'ai ajouté une contrainte `UNIQUE` sur le `timestamp` des prix et le `title` des actualités. Cette simple contrainte empêche d'insérer deux fois la même information, ce qui rend mes scripts de collecte réutilisables à l'infini sans créer de bazar dans la base.

Extrait du Schéma SQL que j'ai défini dans `scripts/stockage.py`.

```
-- Table pour les prix
CREATE TABLE IF NOT EXISTS bitcoin_prices (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp INTEGER UNIQUE, -- La contrainte qui évite les doublons
    open REAL,
    high REAL,
    low REAL,
    close REAL,
    volume REAL
);

-- Table pour les actualités
CREATE TABLE IF NOT EXISTS bitcoin_news (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL UNIQUE, -- La même contrainte ici
    link TEXT NOT NULL,
    content TEXT,
    timestamp DATETIME
);
```

C5 : Rendre les Données Accessibles avec une API

Avoir des données, c'est bien. Pouvoir les utiliser, c'est mieux. Pour ça, j'ai développé une API avec **FastAPI**. J'ai choisi ce framework pour deux raisons principales : il est extrêmement rapide, et surtout, il crée une documentation interactive tout seul.

C'est un gain de temps énorme. N'importe qui peut se connecter à l'URL `/docs` de mon API et voir immédiatement tous les points d'accès disponibles (endpoints), ce qu'ils attendent comme paramètres, et

même les tester en direct. C'est la preuve que l'API est non seulement fonctionnelle, mais aussi facile à utiliser pour d'autres développeurs.

Figure 1 : L'interface Swagger UI générée par **FastAPI**. On y voit les endpoints et on peut les tester directement, ce qui est une preuve concrète d'une API bien documentée. [ACTION REQUISE : Insérez ici votre capture d'écran de l'interface Swagger UI. La capture de la page 13 de votre PDF (Figure 1) est parfaite.]

Conclusion de l'Épreuve E1

À ce stade, la fondation du projet est posée et elle est solide. Les données sont collectées automatiquement depuis plusieurs sources, stockées proprement dans une base de données structurée, et sont prêtes à être utilisées via une API simple et bien documentée. La prochaine étape, la plus intéressante, est de donner vie à ces données avec l'intelligence artificielle.