Explicarea pe bază de prompt engineering a explicațiilor date de SHAP, LIME sau alte astfel de metode

Partea II – Arhitectură și implementare a prototipului

Arădoaie Ioana-Maria, Toader Ana-Maria, Mereu Ioan-Flaviu

1 Scopul etapei curente

În această etapă am implementat un prim prototip funcțional al framework-ului XAI-NLG descris în Partea I. Obiectivul a fost să obținem un *pipeline end-to-end* care:

- pornește de la un model de clasificare (ex. Random Forest) antrenat pe un set de date real;
- extrage contribuții locale ale trăsăturilor cu ajutorul unei metode XAI (în prezent SHAP [1], LIME [2]);
- normalizează contribuțiile și le mapează în enunțuri simple în limbaj natural;
- trece printr-un strat de NLG bazat pe prompt engineering (Few-Shot Prompting [3], Chain-of-Thought [4] și Self-Consistency [5]);
- validează explicațiile generate (sumă SHAP, claritate) și loghează o evidență structurată într-un fișier CSV.

Etapa curentă nu urmărește încă optimizarea completă a prompturilor sau suportul pentru toate seturile de date propuse, ci verificarea că arhitectura propusă poate fi implementată într-un cod clar, modular, ușor de extins.

2 Structura proiectului

Structura proiectului este organizată pe straturi, astfel încât să reflecte arhitectura conceptuală descrisă în Partea I:

- `config/`: centralizează toți parametri într-un singur `FrameworkConfig` (`ExplainerConfig`, `NormalizerConfig`, `NLGConfig`, `ValidatorConfig`).
- `src/explainer/`: implementează stratul XAI (în prezent SHAP și LIME).
- `src/normalizer/`: conține `Normalizer` și `FeatureMapper` pentru scalare și mapare trăsături → enunț.
- `src/nlg/`: implementează generatorul Few-Shot, CoT, Self-Consistency și infrastructura pentru integrarea cu un LLM (folosind Ollama[1]).

[1] https://docs.ollama.com/

- `src/validator/`: conține `Validator` și `EvidenceTracker`, responsabile de verificări și logarea evidențelor.
- `src/pipeline.py`: definește `XAINLGPipeline`, care orchestrează toate straturile.
- `examples/`: scripturi de demo; în această etapă este folosit `breast_cancer_example.py`.

2.1 Configurații și obiectul `FrameworkConfig`

Fișierul `config/settings.py` definește un set de clase pentru configurarea unitară a framework-ului:

- **`ExplainerConfig`**: stabilește metoda de explicare (`"shap"` sau `"lime"`), parametrii specifici (ex. număr de eșantioane pentru KernelSHAP, tipul modelului etc.).
- **`NormalizerConfig`**: controlează metoda de scalare (`"minmax"`, `"standard"` etc.), pragul de grupare a trăsăturilor mici (`feature_grouping_threshold`) și numărul maxim de trăsături de raportat.
- **`NLGConfig`**: conține setările pentru generatorul de text (tehnică `"few_shot"`, model LLM, șabloane).
- **`ValidatorConfig`**: activează/dezactivează verificări (conservarea sumei pentru SHAP, calcul de claritate etc.).
- **`FrameworkConfig`**: agregă toate aceste configurări și expune o interfață unică folosită de `XAINLGPipeline`.

Această structură permite modificarea comportamentului framework-ului (de exemplu trecerea de la SHAP la LIME sau modificarea metodei de normalizare) fără a altera codul de bază.

3 Stratul 1: Explainer (SHAP/LIME)

Stratul de explicare este implementat în `src/explainer/`. Există o interfață comună `BaseExplainer` și două implementări:

- **`ShapExplainer`**: folosește biblioteca SHAP[2] pentru a calcula contribuții locale. În cazul modelului de tip Random Forest utilizat în demo, este instanțiat un `TreeExplainer`; funcția principală expune o metodă de tip:

$$\texttt{explain\_instance}(x) \rightarrow \{\text{feature} \mapsto \phi_{\text{SHAP}}\}.$$

- **`LimeExplainer`**: folosește biblioteca LIME[3], folosind un `LimeTabularExplainer`. Metoda returnează o listă de perechi $(\text{feature}, \text{contribuție})$ care poate fi convertită în același format dicționar.

Explainer-ul primește modelul antrenat și datele de antrenare și produce, pentru fiecare instanță, un dicționar $\{\texttt{feature\_name} \mapsto \texttt{valoare}\}$ ce reprezintă contribuții pozitive sau negative la predicție.

[2] https://shap.readthedocs.io/en/latest/
[3] https://lime-ml.readthedocs.io/en/latest/

4 Stratul 2: Normalizer & Mapper

4.1 Normalizarea contribuțiilor

Clasa `Normalizer` (`src/normalizer/normalizer.py`) implementează trei pași principali:

1. **Normalizare numerică** prin metoda `normalize_contributions`, folosind utilitarul `normalize_values`:
   - *Min-Max scaling*: mapare în intervalul $[0, 1]$, păstrând semnul separat;
   - alte metode (ex. standardizare) pot fi adăugate fără a modifica restul pipeline-ului.
2. **Grupare de trăsături mici** prin `group_small_features`: toate contribuțiile cu modul sub un prag sunt agregate într-o categorie `"others"`. Astfel, raportarea către utilizator se concentrează pe trăsături dominante, iar „coada" este sumarizată.
3. **Ordine descrescătoare** prin `rank_features`: trăsăturile sunt sortate descrescător după valoarea absolută a contribuției, rezultând o listă de tuple $(\text{feature}, \text{valoare})$. Această listă este folosită ulterior atât pentru generarea de text, cât și pentru metricele top-k.

4.2 Maparea trăsăturilor în enunțuri

Clasa `FeatureMapper` (`src/normalizer/mapper.py`) realizează tranziția dintre valori numerice și enunțuri scurte în limbaj natural:

- definește o structură `FeatureStatement` cu câmpuri precum: `feature`, `value`, `rank`, `direction`, `statement`;
- stabilește *direcția* contribuției prin `determine_direction`: pozitivă, negativă sau neutră (în funcție de un prag mic, de exemplu $|v| < 0.01$);
- clasifică *magnitudinea* valorii normalizate (`high`, `medium`, `low`) prin `determine_magnitude`.

Pe baza acestor informații, `map_feature_to_statement` alege o șablonare adecvată, de forma:

- pentru contribuții pozitive mari: `"The {feature} (value: {value:.2f}) strongly supports the prediction."`;
- pentru contribuții negative mari: `"... strongly contradicts the prediction."`;
- pentru contribuții medii sau mici: variante „moderately" / „slightly".

Funcția `map_features` aplică această mapare pe lista trăsăturilor ordonate, rezultând o listă de `FeatureStatement`. În demo, aceste enunțuri sunt afișate în consolă și vor fi, în etapele următoare, comunicate modelului de NLG ca exemple few-shot.

5 Stratul 3: NLG cu Few-Shot Prompting

Stratul NLG este implementat în `src/nlg/few_shot_generator.py` prin clasa `FewShotGenerator`. Rolul ei este să transforme un context structurat într-o explicație finală,

per instanță.

5.1 Contextul furnizat NLG-ului

Pipeline-ul construiește un `context` care conține:

- predicția numerică a modelului și eticheta (de ex. malign / benign);
- lista de trăsături ordonate şi enunțurile scurte generate de `FeatureMapper`;
- informații despre metoda XAI utilizată (SHAP/LIME) și despre modul de normalizare;
- eventuale metrici de verificare (statusul conservării sumei, scor de claritate).

Acest context este apoi trecut către `FewShotGenerator` care:

1. construiește un prompt few-shot prin `build_few_shot_prompt`, inserând exemple de tipul:

   > „Dacă primele trăsături sunt X, Y, Z cu contribuții pozitive mari, generează o explicație concisă care afirmă că acestea susțin puternic predicția."

2. oferă posibilitatea unei versiuni duale (`build_dual_prompt`) pentru a separa explicit partea de analiză de răspunsul final (util pentru Chain-of-Thought).
3. apelează un LLM printr-o funcție dedicată (de exemplu `ollama_call`) sau printr-o funcție injectată din exterior (`llm_call_fn`). În lipsa unei integrări reale, există un fallback care generează o explicație simplificată, dar deterministă.

În etapa curentă, am implementat integrarea cu Ollama, respectând separarea între logica pipeline-ului și logica de apel la modelul de limbaj.

6 Stratul 4: Validator & evidență

6.1 Verificări automatizate

Clasa `Validator` (`src/validator/validator.py`) implementează două tipuri principale de verificări:

- **Conservarea sumei SHAP**: pentru modele arbori, se verifică dacă

$$\sum_i \phi_i \approx F(x) - \mathbb{E}[F],$$

  în limita unei toleranțe numerice. Rezultatul este un flag `sum_conservation_valid` și o valoare numerică `computed_sum`.

- **Claritatea explicației**: este calculat un scor simplu de claritate (ex. pe o scară 0–100), bazat pe proprietăți elementare ale textului (lungime, repetitivitate, prezență de termeni cheie). În etapele viitoare, acesta poate fi înlocuit sau completat cu evaluări umane și metrice mai sofisticate.

Rezultatul validatorului este un dicționar ce capturează starea acestor verificări, folosit atât pentru raportare, cât și pentru logarea evidențelor.

6.2 Evidențe și export CSV

Clasa `EvidenceTracker` (`src/validator/evidence_tracker.py`) păstrează legătura dintre text și valorile XAI:

- definește structura `EvidenceRecord`, cu câmpuri precum: `timestamp`, `statement`, `method`, `features`, `contributions`, `fidelity_score`, `validation_status`;
- oferă o metodă `add_record` pentru a înregistra, pentru fiecare explicație generată, trăsăturile corespunzătoare și verificările asociate;
- poate exporta evidențele în format CSV prin `export_csv`, rezultând un fișier de tip `evidence_audit.csv` ce poate fi analizat ulterior (ex. în Excel sau cu pandas).

În demo-ul curent, după rularea pipeline-ului pe câteva instanțe, scriptul `examples/breast_cancer_example.py` apelează exportul CSV, creând un audit minimal al legăturilor „explicație textuală ↔ trăsături ↔ verificări”.

7 Pipeline-ul end-to-end

Clasa centrală `XAINLGPipeline` (`src/pipeline.py`) leagă toate componentele descrise anterior. Metoda principală, `explain_instance`, urmează pașii:

1. **Explainer**: calculează contribuțiile trăsăturilor pentru instanța analizată folosind SHAP sau LIME, în funcție de configurație.
2. **Normalizare**: aplică `Normalizer` pentru a obține valori scalate, eventual grupate, și o listă de trăsături ordonate.
3. **Mapare în enunțuri**: `FeatureMapper` transformă aceste trăsături într-o listă de enunțuri scurte.
4. **Construcția contextului**: pipeline-ul construiește un obiect de context cu predicția numerică, eticheta, trăsăturile top-k, enunțurile mapate și meta-informațiile XAI.
5. **Generare NLG**: un generator (`FewShotGenerator`, `ChainOfThoughtGenerator` sau SelfConsistencyGenerator) este apelat cu acest context și produce o explicație în limbaj natural.
6. **Validare**: `Validator` verifică conservarea sumei SHAP (dacă este cazul) și calculează scorul de claritate, iar `EvidenceTracker` înregistrează rezultatul.
7. **Rezultat unificat**: toate aceste informații sunt ambalate într-o structură care conține, printre altele:
   - predicția modelului (`prediction`);
   - explicația generată (`generated_text`);
   - detalii despre trăsături (`feature_statements`);
   - rezultate de validare (`validation_summary`).

Metoda `explain_batch` extinde această logică pentru colecții de instanțe, fiind utilă pentru evaluări pe subseturi de test sau pentru generarea de rapoarte agregate.

8 Exemplu pe setul de date Breast Cancer

Scriptul `examples/breast_cancer_example.py` instanțiază framework-ul pe dataset-ul Breast Cancer Wisconsin[4]:

- încarcă datele (prin `sklearn.datasets`[5]);
- împarte datele în antrenare și test;
- antrenează un model de clasificare (Random Forest);
- construiește un obiect `FrameworkConfig` cu parametri expliciți (ex. `explainer.method="shap"`, `normalizer.feature_grouping_threshold` etc.);
- creează o instanță `XAINLGPipeline` și aplică `explain_instance` pentru o instanță de test.

Rularea scriptului tipărește în consolă:

- predicția numerică (`Prediction: 1`);
- explicația generată;
- lista trăsăturilor top-k și enunțurile corespondente, de forma:
  `worst area 0.07 -> The worst area (value: 0.07) strongly supports the prediction.`
- un rezumat al verificărilor, de tip:

```
{
'sum_conservation_valid': False,
'computed_sum': 0.97,
'clarity_score': 78.0
}
```

Aceste rezultate confirmă că pipeline-ul reușește să conecteze contribuțiile XAI cu un text generat și cu verificări elementare. Diferența mică la suma SHAP indică posibile surse de eroare numerică sau necesitatea ajustării toleranței, element ce poate fi discutat în secțiunea de limitări.

9 Limitări și pași viitori

Etapa curentă livrează un prototip funcțional, dar cu limitări intenționate:

- **Metode XAI**: în demo este utilizat SHAP pentru un model de tip arbori; suportul LIME este conturat, dar nu evaluat riguros.

[4] https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data
[5] https://scikit-learn.org/stable/api/sklearn.datasets.html

- **NLG**: generatorul few-shot folosește în prezent un fallback simplificat; integrarea efectivă cu Ollama (sau alt LLM) reprezintă următorul pas logic.
- **Prompt engineering**: sunt pregătite hook-uri pentru tehnici de tip Chain-of-Thought și Self-Consistency, însă pipeline-ul curent folosește doar un fir simplu de generare.
- **Evaluare**: scorul de claritate este rudimentar; nu există încă studii cu utilizatori umani și nici metrice avansate inspirate din literatura de prompt engineering [6].
- **Seturi de date**: în această etapă este integrat complet doar setul Breast Cancer Wisconsin[6]; pentru Adult Income[7] vor fi necesari pași suplimentari de preprocesare și tratare a trăsăturilor categoriale.

Pașii viitori includ:

- testarea mai multor modele LLM;
- extinderea stratului de validare (stabilitate sub perturbări, Jaccard@k, metrice de consistență temporală);
- adăugarea de șabloane multi-lingve (română/engleză) în `FeatureMapper`;
- evaluarea cu utilizatori pentru a calibra lungimea, stilul și utilitatea explicațiilor generate.

Prin această etapă, framework-ul propus trece de la o arhitectură teoretică la un prototip concret, capabil să ruleze de la un model ML antrenat până la o explicație textuală validată și logată, păstrând trasabilitatea dintre valori XAI și textul generat.

Bibliografie

[1] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions, 2017.

[2] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier, 2016.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[6] https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data
[7] https://www.kaggle.com/datasets/wenruliu/adult-income-dataset

[5] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[6] Shubham Vatsal and Harsh Dubey. A survey of prompt engineering methods in large language models for different nlp tasks, 2024.