Adattárházak, adatbányászati technológiák EA projektfeladat: Automatikus kódtranszformáció BLOOM modell segítségével

1st Nick Mónika *Információs Rendszerek*
ELTE, Eötvös Loránd Tudományegyetem
Budapest, Hungary
uhh5ks@inf.elte.hu

2nd Jurca Henriette *Információs Rendszerek*
ELTE, Eötvös Loránd Tudományegyetem
Budapest, Hungary
sj47tr@inf.elte.hu

3rd Amamou Martin *Szoftvertechnológia*
ELTE, Eötvös Loránd Tudományegyetem
Budapest, Hungary
amamoumartin@gmail.com

2025. július 16.

Kivonat

Ez a projektfeladat a BLOOM nagy nyelvi modell alkalmazását vizsgálja automatikus kódtranszformáció területén. Kísérleteink során értékeltük a modell hatékonyságát Python-Java és C++-Java kódkonverziók végrehajtásában, különös figyelmet fordítva a szintaktikai pontosságra és szemantikai megőrzésre. Eredményeink azt mutatják, hogy a BLOOM - bár nem kifejezetten kódgenerálásra lett kiképezve - ígéretes teljesítményt nyújt többnyelvű programozási nyelvi transzformációk terén.

1. Bevezetés

A nagy nyelvi modellek (LLM) gyors fejlődése forradalmasította a programozási nyelvek közötti automatikus transzformáció területét. Jelen tanulmány célja a BLOOM nyílt forráskódú modell képességeinek vizsgálata ezen a területen, különös tekintettel a Python-Java és C++-Java kódátalakításokra.

A BLOOM modell választását több tényező indokolja:

- Nyílt forráskódú természet, amely lehetővé teszi a teljes körű testreszabást és transzparens használatot

- Többnyelvű képességek, beleértve a Python, C++ és Java támogatását
- Nagy kontextusablak (akár 2048 token), amely lehetővé teszi összetett kódszegmensek feldolgozását
- A BigScience kezdeményezés eredményeképpen kialakított, átlátható adatbázis és képzési folyamat

2. A BLOOM modell részletes bemutatása

2.1. A modellről

A BLOOM (BigScience Large Open-science Open-access Multilingual) [1] egy 176 milliárd paraméterű, decoder-only transformer architektúrájú modell, amely technikai szempontból hasonlít a GPT-3-hoz, de kifejezetten többnyelvű feladatokra lett kiképezve. A modellen többezer tudós, fejlesztő dolgozott, és óriási mennyiségű adattal lett betanítva: 43 emberi nyelv, 13 féle programozási nyelv, és nagyjából 1.6 TB-nyi előre feldolgozott szöveges adattal. [2] A projektet kezdetben a **Huggingface** társalapítója kezdte el, de később egyéb cégek is részt vettek benne, mint például a **Microsoft**, az **NVIDIA**, vagy a **PyTorch**.

2.2. Technikai háttér

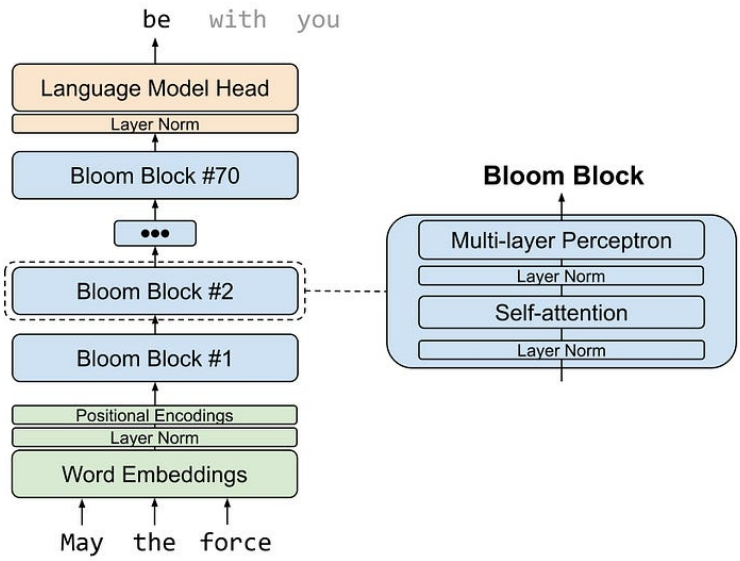
A BLOOM modell legfőbb célja egy adott szövegben a következő tokenek prediktálása, az előzők halmaza alapján. Ez a stratégia bizonyította, hogy egy adott szintű érvelési képességet ér el LLM-ek esetében. Emiatt képes a Bloom, és a többi hasonló modell arra, hogy fogalmakat kapcsoljanak össze mondatokban, továbba nem triviális feladatokat, matematikai, programozási problémákat oldjanak meg, megfelelő pontossággal.

A Bloom modell a fenti ábrán látható, ún. Transformer architektúrát használja [3], amely az alábbi komponensekből áll:

- Bemeneti beágyazó réteg (Input Embedding)
- 70 Transformer dekóder blokk
- Kimeneti nyelvi modellező réteg (Language Model Head)

Minden ilyen Transformer blokk az alábbiakból épül fel:

1. **Önfigyelő réteg (Self-Attention)**:
    - Multi-head attention mechanizmus
    - 16 figyelő fej a BLOOM-560M esetén
    - Causal masking (megakadályozza a "jövőbeli" tokenek láthatóságát)



1. ábra. A BLOOM modell architektúrája

2. **Többrétegű perceptron (MLP)**:
   - 4-szeres rejtett dimenzió (560M modellnél 1024 → 4096)
   - GeLU aktivációs függvény

Ahhoz, hogy a BLOOM segítségével megjósoljunk egy mondat következő tokenjét, az input tokeneket át kell adnunk az összes blokkon keresztül.

A kísérletek során a `bigscience/bloom-560m` verziót alkalmaztuk, amely a modellcsalád legkisebb méretű tagja. A verzió előnye, hogy különösen alkalmas kisebb erőforrásokkal rendelkező környezetekben való kísérletezésre és fejlesztésre.

2.3. Telepítés és használat

A BLOOM-560M modell használata meglepően egyszerű a Hugging Face `transformers` könyvtárának segítségével. Az alábbi példával bemutatjuk a modell telepítésének és betöltésének lépéseit.

2.3.1. Előfeltételek

- Python 3.7 vagy újabb
- `transformers` könyvtár (4.21.0 vagy újabb)
- `torch` vagy `tensorflow` backend
- Legalább 3GB RAM (CPU) vagy 2GB GPU memória

2.3.2. Telepítés lépései

1. Könyvtárak telepítése:

```
pip install torch transformers
```

2. A modell első használatakor automatikusan letöltődik (1.2GB)

3. Használat előtt érdemes ellenőrizni a rendelkezésre álló erőforrásokat

```
from transformers import (AutoModelForCausalLM, AutoTokenizer)

model_id = "bigscience/bloom-560m"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)
```

Listing 1. BLOOM-560M betöltése

A kód magyarázata:

- `AutoModelForCausalLM`: A Hugging Face által biztosított osztály generatív nyelvmodellek betöltéséhez.
- `AutoTokenizer`: A modellhez tartozó tokenizáló betöltése.
- `model_id`: A modell azonosítója a Hugging Face Model Hub-on.
- `from_pretrained`: Automatikusan letölti és betölti a modellt és tokenizálót.

2.4. Hardverigény

A BLOOM modell eredeti képzése a franciaországi Jean Zay szuperszámítógépen történt NVIDIA A100 GPU-k felhasználásával. A modell használata azonban jelentős memóriaigényt támaszt:

1. táblázat. BLOOM modellváltozatok erőforrás-igénye

Modellváltozat	Paraméterek	GPU RAM követelmény
bloom-1b7	1.7B	4GB
bloom-3b	3B	8GB
bloom-7b1	7B	16GB
bloom-176b	176B	>350GB

2.5. Limitációk

A BLOOM modell használata során figyelembe kell venni a következő korlátozásokat [2] és kockázatokat:

2.5.1. Reprezentációs torzítások

- Egyes nézőpontok **túlreprezentálása**, míg mások **alulreprezentálása**
- Sztereotípiákat tartalmazhat a tanulási adatokból örökölve
- Véletlenül feldolgozhat és megjeleníthet **személyes információkat**

2.5.2. Tartalmi kockázatok

A modell esetleg generálhat:

- **Gyűlölködő**, gyalázkodó vagy erőszakos nyelvezetet
- **Diszkriminatív** vagy előítéletes kifejezéseket
- Környezetfüggően **megfelelőtlen tartalmakat**, ideértve:
  - Szexuális utalásokat
  - Erőszakos tartalmakat

2.5.3. Műszaki korlátozások

- **Téves információk** tényként való prezentálása
- **Ismétlődő** vagy irreleváns kimenetek generálása
- Strukturális hibák a következő területeken:
  - Ténybeli pontatlanságok
  - Logikai inkonzisztenciák

2.5.4. Antropomorfizálási veszély

- A felhasználók hajlamosak **emberi tulajdonságokat** tulajdonítani a modellnek:
  - Érzékenység
  - Tudatosság
  - Szándékosság

3. Módszertan

3.0.1. Fájlstruktúra

A rendszer az alábbi mappastruktúrát és adatfájlokat használja:

```
CodeConv/
           cpp-java.jsonl
           cpp_test_files.jsonl
           cpp-reader.py
           cpp-java-learn.py
           cpp-java-test.py
           cpp-test-files/
                 01.cpp
                 02.cpp
                 ...
           java-files/
                 01.java
                 02.java
                 ...
           java-files-corrected/
               01_ok.java
               02_ok.java
               ...
```

Listing 2. Fájlstruktúra

3.0.2. Működési folyamat

A konverziós folyamat lépései a következők:

- `cpp-reader.py`
  Beolvassa a `cpp-test-files` mappában található C++ fájlokat (57 db), és létrehozza a `cpp_test_files.jsonl` fájlt.

- `cpp-java-learn.py`
  A BLOOM modellt betanítja C++–Java párokon a `cpp-java.jsonl` fájl alapján.

- `cpp-java-test.py`
  A betanított modell segítségével Java-kódokat generál a `cpp_test_files.jsonl` állományból, majd elmenti azokat a `java-files` mappába.

A Python-Java konverzió hasonló módon működik, ahol a `py-java.jsonl` fájl tartalmazza a betanuláshoz szükséges adatpárokat.

3.1. A modell betanítása

Az alábbiakban, a modellt tanító `cpp-java-learn.py` szkript működését mutatjuk be:

A szkript működése a következő lépésekre bontható:

1. **Modell és tokenizer betöltése**
   A `transformers` könyvtár segítségével a BLOOM-560m nyelvi modellt és a hozzá tartozó tokenizert inicializáljuk.

2. **Tanító adatok beolvasása**
   A `cpp-java.jsonl` fájl minden sora egy `input` (C++) és egy `output` (Java) kódrészletet tartalmaz.

3. **Előfeldolgozás**
   A modellt egy strukturált `.jsonl` fájllal tanítjuk. Minden sor egy objektumot tartalmaz `input` és `output` kulcsokkal, ahol:

   - `input`: a C++ nyelvű kód,
   - `output`: a hozzá tartozó Java-kód.

   A modell számára a tanításhoz szükséges promptot ezekből az elemekből állítjuk elő a betöltéskor.

   A bemeneti példák a következő formátumban szerepelnek az adatfájlban:

```
{
  "input": "int add(int a, int b) { return a + b; }",
  "output": "public int add(int a, int b) { return a + b;
      }"
}
```

Listing 3. Bemeneti JSON példa

   A generált Java kimenetek végét a jel határolja, így a szkript egyszerűen ki tudja vágni a hasznos kódrészt.

4. **Tokenizálás**
   A kódrészleteket 512 token hosszúságig vágjuk, és batch-ben előkészítjük a tanításhoz.

5. **Fine-tuning beállítások**
   A modell három epizódon keresztül tanul, kis batch mérettel, GPU támogatással (ha elérhető). A tanítás során minden epoch végén mentésre kerül az aktuális modellállapot.

6. **Tanítás és mentés**
   A `Trainer` osztály végzi el a tényleges tanítást. A tanítás végén a modell és a tokenizer is elmentésre kerül a `./bloom-cpp2java` mappába.

3.2. A modell tesztelése

A cpp-java-test.py működése lépésről lépésre:

1. **Finomhangolt modell betöltése**
   A `./bloom-cpp2java` könyvtárból betöltésre kerül a korábban tanított BLOOM modell és tokenizáló, majd egy `text-generation` típusú `pipeline` objektum jön létre.

2. **Bemeneti fájl és output mappa definiálása**
   A szkript a `cpp_test_files.jsonl` állományt olvassa soronként, és az elkészült Java kódokat a `java-files` mappába menti el. A könyvtár létrehozása automatikusan történik, ha nem létezik.

3. **Bemeneti fájl feldolgozása**
   A JSONL fájl minden sorában megkeresi a C++ fájl nevét és tartalmát. Az üres vagy hiányos bejegyzéseket kihagyja.

4. **Prompt összeállítása**
   Minden bemeneti C++ kódhoz elkészül a konzisztens prompt formátum:

   ```
   C++:\n{cpp_code}\n\nJava:\n
   ```

   A modell ez alapján folytatja a kódot Java nyelven.

5. **Java kód generálása**
   A BLOOM modell a megadott prompt alapján legenerálja a Java nyelvű kódváltozatot. A válaszból a szkript eltávolítja az eredeti promptot, és a határolóig tartó szöveget tekinti a releváns generált kódnak.

   A generálás során a következő paramétereket alkalmaztuk:

   - **prompt**: A bemeneti szöveg, amely tartalmazza a `C++:` kulcsszót, a C++ kódrészletet, majd egy `Java:` részt, amely után a modellnek folytatnia kell a Java megfelelőjével.
   - **max_length**: A generált szekvencia maximális hossza, amelyet a bemenet hosszának és egy fix határnak az összevetésével határoztunk meg (`min(input_length + 1024, 2048)`).

- **do_sample=True**: Mintavételezés engedélyezése a deterministikus válaszok helyett, így változatosabb, alternatív kimenetek is generálhatók.
- **temperature=0.7**: A mintavételezési hőmérséklet beállítása, amely befolyásolja a modell válaszainak kreativitását – az alacsonyabb érték konzervatívabb, míg a magasabb változatosabb válaszokat eredményez.

6. **Java fájl mentése**
   A generált kimenet a bemeneti fájl nevéből képzett `.java` kiterjesztésű fájlba kerül a `java-files` mappába. A könyvtárszerkezet automatikusan létrejön szükség esetén.

7. **Hibakezelés**
   Ha a feldolgozás során hiba lép fel (pl. JSON hiba, tokenizálási probléma), a szkript naplózza a hibát, de a feldolgozást folytatja a következő bejegyzéssel.

A szkript előnye, hogy teljesen automatikusan képes nagyszámú fájlon végrehajtani a konverziót, így kiválóan alkalmas batch-teszteléshez és tömeges elemzéshez.

3.3. Adathalmaz

A kísérletekhez létrehozott adathalmazunk, a következő típusú kódszegmenseket tartalmazza:

- Egyszerű függvények (pl. faktoriális, Fibonacci)
- Vezérlési szerkezetek (for, while, foreach, switch, if)
- Osztálydefiníciók és öröklődés
- Alapvető adatszerkezetek (tömbök, listák, map)
- Egyszerűbb algoritmusok (min/maxkiválasztás)
- Sztringműveletek
- Fájlból beolvasás, és kiírás
- Struct, enum

3.4. Értékelési metrikák

A modell teljesítményét manuálisan értékeltük 48 tesztpélda alapján. A generált Java kódokat összevetettük az elvárt kimenetekkel, és 34 esetben megfelelő eredményt adott, míg 14 esetben hibás volt a kimenet.

Ez alapján a modell pontossága:

$$\text{Pontosság} = \frac{34}{48} \approx 70{,}83\%$$

4. Eredmények és Elemzés

4.1. Példák generált kódra

A modell számos esetben képes volt helyesen transzformálni az alapvető programozási konstrukciókat:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Listing 4. Python input

```
public static int factorial(int n) {
    return n == 0 ? 1 : n * factorial(n-1);
}
```

Listing 5. Java output

4.2. Hibák típusai

A gyakorlatban előforduló hibák főbb kategóriái:

- Szintaktikai hibák (elfelejtett pontosvessző, zárójelek)
- Nyelvspecifikus konstrukciók helytelen leképezése
- Memóriakezelési különbségek (pl. C++ pointerek Java-ban)
- Hiányzó fordítási részek: a generált Java kód nem tartalmazza a teljes konverziót
- Stílusbeli vagy nem idiomatikus Java: bár nem hiba, de a Java-kód nem követi a szokásos konvenciókat

5. További munkák és bővítési lehetőségek

A jelen projekt során a `bloom-560m` verziót használtuk, amely méretében és memóriaigényében jól illeszkedik közepes erőforrású környezetekhez (például

Google Colab Pro vagy otthoni GPU-s gépek). Azonban a modell teljesítménye, pontossága és általánosíthatósága tovább javítható az alábbi fejlesztési lehetőségekkel:

1. **Nagyobb BLOOM modellek alkalmazása**

   A nagyobb méretű BLOOM modellek (például `bloom-1b7`, `bloom-3b`, `bloom-7b1`, vagy akár a `bloom-176b`) képesek komplexebb kódminták és szemantikai összefüggések pontosabb feldolgozására. Használatuk jelentősen javíthatja a kódtranszformáció szintaktikai pontosságát és szemantikai megőrzését.

   Előnyeik:

   - Mélyebb nyelvi kontextus megértése
   - Összetettebb konstrukciók kezelése (például öröklődés, generikus típusok)
   - Kevesebb hiba a nyelvspecifikus eltérések miatt

   Szükséges erőforrások:

   - `bloom-1b7` modellhez legalább 4 GB GPU memória
   - `bloom-3b` modellhez 8 GB GPU RAM
   - `bloom-7b1` modellhez legalább 16 GB GPU RAM (pl. NVIDIA RTX A5000 vagy Tesla T4 / A100)
   - `bloom-176b` modell futtatása csak nagyobb klasztereken vagy felhőszolgáltatásokon (pl. AWS, Azure, Hugging Face Inference Endpoints) lehetséges

   Hol és hogyan?
   A Hugging Face `transformers` könyvtárán keresztül a nagyobb modellek is betölthetők, de jellemzően nem Colab-on, hanem dedikált gépeken vagy bérelt GPU-s szervereken (pl. Vast.ai, LambdaLabs, RunPod) célszerű őket futtatni.

2. **Finomhangolás nagyobb, kódra specializált adathalmazokon**

   A nagyobb BLOOM modellek jobban finomhangolhatók domain- specifikus feladatokra. A `CodeXGlue`, `CodeNet`, vagy GitHub-alapú kódpárok segítségével tovább javítható a konverziós képesség.

3. **Modellváltozatok összehasonlítása**

   A BLOOM mellett más nagy nyelvi modellek is kipróbálhatók ugyanazon feladaton (például `CodeGen`, `StarCoder`, `LLaMA`, `CodeT5+`), és az eredmények összevethetők pontosság, sebesség és erőforrásigény szempontjából.

4. **Kódhibák automatikus javítása**

   A nagyobb modellek képesek lehetnek nem csak konvertálni, hanem önállóan javítani is a generált kódot (például hiányzó pontosvessző, zárójelezési hiba, hibás típuskonverzió). Ezáltal a programozói munka tovább csökkenthető.

5. **Egységtesztek automatikus generálása**

   A generált Java kód mellé automatikusan létrehozhatók lennének `JUnit` tesztek, amelyek ellenőrzik, hogy a konvertált kód szemantikailag megfelel-e az eredetinek. Ez növelné az automatizált validáció lehetőségét.

6. **Többirányú transzformáció**

   Jelenleg a rendszer egyirányú transzformációt végez (például Python → Java). Bővítési lehetőség a kétirányú (bidirekcionális) konverzió megvalósítása, így ugyanaz a modell képes lenne Java → Python átalakításra is.

7. **Interaktív felhasználói felület fejlesztése**

   A rendszer integrálható lenne egy grafikus vagy webes felületre, ahol a felhasználó feltölthet kódot, választhat célnyelvet, és azonnal megkapja a transzformált kódot. Ez lehetővé tenné nem technikai felhasználók számára is a szolgáltatás kihasználását.

Hivatkozások

[1] A. Suvorov. "BLOOM: An Open Multilingual Language Model". Online tutorial article. (2023. jan.), cím: `https://medium.com/@alexander.suvorov/overview-a6ce99a827c4` (elérés dátuma 2025. 05. 06.).

[2] BigScience. "BLOOM (BigScience Large Open-science Open-access Multilingual)". Online tutorial article. (2022. júl.), cím: `https://huggingface.co/bigscience/bloom` (elérés dátuma 2025. 05. 06.).

[3] D. S. Team. "Run BLOOM — The Largest Open-Access AI Model on Your Desktop Computer". Online tutorial article. (2023. márc.), cím: `https://medium.com/data-science/run-bloom-the-largest-open-access-ai-model-on-your-desktop-computer-f48e1e2a9a32` (elérés dátuma 2025. 05. 06.).