LLM ile Kod Çevirisi için In-Context Learning

1. Deneysel Çalışma (Experimental Study)

Bu proje, Büyük Dil Modellerinin (LLM'ler) farklı *prompting setups* altında **code translation** (kod çevirisi) konusunda nasıl performans gösterdiğini analiz etmeyi amaçlamaktadır: **zero-shot (sıfır örnekli), few-shot (az örnekli) ve many-shot (çok örnekli) prompting.** Birden fazla LLM'yi karşılaştıracağız ve örnek sayısını artırmanın çeviri kalitesini iyileştirip iyileştirmediğini araştıracağız.

2. Metodoloji (Methodology)

Zero-shot, few-shot ve many-shot prompting yöntemlerini karşılaştırmak için öncelikle bir veri setine ihtiyacımız var. Bu veri setinden bir kısmını **test veri seti** olarak ayırmamız gerekiyor ve geri kalanını **training veri seti** (eğitim veri seti) olarak kullanacağız. Training veri setinden ise few-shot ve many-shot promptlarımız için örnekler (input-output ikilileri) hazırlamamız gerekecek. Bu şekilde LLM'leri Test veri setimiz üzerinden farklı prompting setup'larda (zero-shot, few-shot, many-shot) değerlendirebiliriz.

2.1. Veri Seti Toplama (Data Collection)

- Öncelikle birkaç programlama dili çifti seçiyoruz (örn. Python → Java, C++ → C).
- **CodeTransOcean** veya aynı algoritmanın kodunun birden fazla dilde mevcut olduğu başka bir güvenilir veri seti buluyoruz. (Size tavsiyemiz CodeTransOcean veri setini kullanmanızdır.)
- Aynı veri setinden, **test dataseti** olarak yaklaşık **~50 girdi** veya buna yakın bir sayıda ayırın (CodeTransOcean'da zaten kullanabileceğiniz ayrı training ve test veri setleri ayrı klasörlerde bulunmaktadır).
- Ayrıca, **training dataseti** olarak da yaklaşık **~500 örnek** ayırın. (Örnek derken şunu kastediyoruz: 500 farklı algoritma düşünün. Her algoritma için hem Python dilinde yazılmış bir kodu, hem de aynı algoritmanın Java dilinde yazılmış kodunu yan yana getirin. İşte bu Python kodu ve Java kodu çifti bizim bir örneğimiz oluyor. Toplamda 500 tane böyle Python-Java kod çiftine ihtiyacımız var.)

2.2. Prompting Setups

Şimdi LLM'leri farklı **prompting setups** altında test edeceğiz:

- **Zero-shot (Sıfır Örnekli):** Prompt içerisinde hiç örnek olmayan kurulum. LLM'ye doğrudan çeviri görevi verilir ve örnek gösterilmez.
- **Few-shot (Az Örnekli):** 5 veya 25 örnek içeren prompt. LLM'ye çeviri görevi ile birlikte birkaç örnek girdi-çıktı çifti verilir.
- **Many-shot (Çok Örnekli):** 125 veya 625 örnek içeren prompt (daha büyük bir örnek kümesi olan prompt). LLM'ye çeviri görevi ile birlikte çok sayıda örnek girdi-çıktı çifti verilir.

  Buradaki örnek sayıları değiştirilebilir. Ancak prompt'un *Many-shot prompt* olarak kabul edilmesi için en az 100 örneğe ihtiyacımız var.

2.3. LLM'leri Test Etme (Testing LLMs)

Şimdi her LLM'yi test datasetindeki her girdi için, zero-shot, few-shot, ve many-shot promptlar ile test edeceğiz.

- Farklı boyutlarda birden fazla LLM kullanın (örn. Gemini Flash 2.0, Pro 2.0, Flash Lite, vb.). Farklı parametre boyutlarına sahip modelleri deneyin. Örneğin, Gemini Pro büyük ve daha maliyetlidir. Flash Lite veya Flash 8b daha küçük ve daha ekonomiktir. Hem küçük hem de büyük modelleri test etmek iyi bir yaklaşımdır.

Burada örneğin eğer 5 farklı prompt setup'ımız varsa (bir zero-shot, iki few-shot, iki many-shot), ve test veri setimizde ise 50 girdi varsa, toplamda 250 defa her LLM'yi çalıştıracağız. LLM'lerin ürettiği çıktıları bir yerde (.txt veya .json veya .csv formatında) kaydediyoruz ve bir sonraki adımda LLM'leri farklı metrikler ile değerlendireceğiz.

3. Değerlendirme (Evaluation)

LLM'lerin başarısını değerlendirmek için **dinamik** ve **statik** metrikler kullanıyoruz.

3.1. Dinamik Metrikler

- **Pass@1**: Çevrilen kodun derlenip doğru şekilde çalışıp çalışmadığını test eder. Örneğin Python → Java yapacaksanız, LLM'in ürettiği çıktı (Java kodu), derlenip derlenmediğini binary olarak test etmek. Eğer kod derleniyorsa başarılı (1), derlenmiyorsa başarısız (0) olarak işaretlenir.
- **Success@1**: Üretilen kodun, verilen bir girdi için beklenen çıktıyı üretip üretmediğini kontrol eder. Örneğin bubble sort algoritmasını Python → Java çevirdiğinizde, LLM'in ürettiği Java kodu verdiğiniz bir diziyi sıralayabiliyor mu? (Kodun algoritmik doğruluğunu test etmek amacıyla). Eğer kod doğru çalışıyorsa ve beklenen çıktıyı üretiyorsa başarılı (1), aksi takdirde başarısız (0) olarak işaretlenir.

3.2. Statik Metrikler

Bu metrikler NLP yaklaşımını kullanarak LLM'in ürettiği kodun ne kadar doğru koda (CodeTransOcean'daki referans koda) benzediğini hesaplarlar.

1. **BLEU**: Çevrilen ve referans kod arasındaki token seviyesindeki örtüşmeyi ölçer. Yüksek BLEU skoru, çevrilen kodun referans koda token seviyesinde ne kadar benzediğini gösterir.
2. **CodeBLEU**: Yapıyı ve mantıksal denkliği dikkate alır. BLEU'ya ek olarak, kodun Abstract Syntax Tree (AST) ve Data Flow Graph (DFG) de dikkate alarak kodun yapısal ve semantik benzerliğini daha iyi yakalar.

Bu metriklerin kendi hazır kütüphaneleri vardır. Projenizde bu kütüphaneleri kullanarak metrik skorlarını otomatik olarak hesaplayabilirsiniz.

4. Uygulama (Implementation)

Bu experimental study'yi Python kodu ile otomatik bir şekilde yapacaksınız. Örneğin Python kodu içerisinde promptlarınız oluşturulacak, promptlara test veri setinden bir test girdisi eklenecek ve zero-shot, few-shot ve many-shot promptlar hazırlanacak ve API call ile LLM'e gönderilecek ve LLM'lerin verdiği cevaplar kaydedilecek.

Daha sonra bu cevaplar seçtiğimiz metrikler ile test edilecek ve sonuçlarımız aşağıdaki örnek tablo gibi bir tabloya eklenecek:

	Pass@1	**Success@1**	**BLEU**	**CodeBLEU**
Zero-shot	x	y	z	k
Few-shot (5)	x	y	z	k
Few-shot (25)	x	y	z	k
Many-shot (125)	x	y	z	k
Many-shot (625)	x	y	z	k

Her model için ayrı bir değerlendirme tablosu olacak. Böylece farklı LLM'lerin ve prompting stratejilerinin performansını karşılaştırabileceksiniz.

---

5. Beklenen Sonuçlar

- *In-context* örneklerini artırmak çeviri kalitesini iyileştiriyor mu? (Örnek sayısı arttıkça çeviri başarımı artıyor mu?)
- Model boyutu, farklı *prompting setups*'lardaki performansı nasıl etkiliyor? (Daha büyük modeller mi, yoksa daha küçük modeller mi farklı prompting yöntemlerinde daha iyi performans gösteriyor?)
- **Prompt length ve model effectiveness** arasındaki *tradeoff* nedir? (Daha uzun promptlar her zaman daha iyi sonuç veriyor mu, yoksa modelin boyutu ve maliyeti ile prompt uzunluğu arasında bir denge kurmak gerekiyor mu?)

Örnek Prompt Şablonu

Örnek bir few-shot prompt şablonu şu şekilde olabilir:

```
# ===== Your Task =====
You are an expert software engineer.
You have to translate code from {source_language} to {target_language}.
Below are some examples input-outputs codes (codes in source and target language) that you can use to learn from and improve your translation ability.
After the example pairs, I am going to provide another code in {source_language} and I want you to translate it into {target_language}.

# ===== Output Structure =====
Return only the translated code in {target_language}.
Do not add any extra commentary, formatting, or chattiness.

# ===== Examples =====
Example 1 (input-output pairs from training dataset)
Example 2 (input-output pairs from training dataset)
...
Example N (input-output pairs from training dataset)

# ====== Code in the {source_language} to be translated to {target_language} =====
{code_in_src_language}
```

- Böyle bir şablon kullanarak promptlarınıza placeholder {} ekleyerek prompt oluşturmayı Python kodu içinde otomatikleştirebilirsiniz.
- Bu format, **tüm prompting setups** genelinde her test verisi için **aynı kalır**, yalnızca **örnek girdi-çıktı çiftlerinin** sayısı değişecek. Örneğin, zero-shot prompting için "Examples" bölümü boş kalacak, few-shot ve many-shot prompting için ise ilgili sayıda örnek girdi-çıktı çifti eklenecektir.
