Büyük Veri Üzerinde Soru Kalitesi Tahmininde ML Boru Hattı Geliştirme - CPU ve GPU İmplementasyonları

github link'i

Ad Soyad: Beren Ünveren
Numara: 221101006

Bu proje Stack Overflow platformundaki soruların kalitesini (Yüksek Kaliteli (HQ), Düşük Kaliteli Düzenlenmiş (LQ_EDIT), Düşük Kaliteli Kapatılmış (LQ_CLOSE)) tahmin etmek amacıyla Apache Spark üzerinde ölçeklenebilir bir ML boru hattı geliştirmeyi hedeflemiştir. Projenin temel amacı bu boru hattını iki farklı GPU hızlandırma stratejisi (NVIDIA RAPIDS'in otomatik entegrasyonu, PySpark Pandas UDF'leri aracılığıyla cuML/cuDF kütüphanelerinin entegrasyonu) ile optimize etmek ve tamamen CPU tabanlı bir referans uygulamasıyla performans ve doğruluk açısından karşılaştırmaktır.

Uçtan uca çözüm; metin ve sayısal özellik mühendisliği, model eğitimi ve tahmin adımlarını içermektedir. Yapılan çalışmalar veri yükleme ve temel DataFrame operasyonlarında GPU hızlandırmasının önemli hızlanmalar sağladığını ancak özellikle özellik mühendisliği aşamasında Spark MLlib'deki bazı bileşenlerin VectorUDT kullanımı gibi mevcut CPU bağımlılıkları nedeniyle darboğazlar oluşturduğunu göstermiştir. Bu darboğazlar beklenen tam GPU ivmelenmesini engellemiş ve bazı durumlarda CPU'dan daha yavaş performans sergilemelerine neden olmuştur. Genel boru hattı hızlanma oranları hesaplanmış ve gözlemlenen darboğazlar detaylandırılmıştır.

1. Giriş

1.1. Problem Tanımı

Stack Overflow gibi geniş çaplı bilgi paylaşım platformlarında yüksek kaliteli içeriklerin hızlıca tanımlanması ve düşük kaliteli içeriklerin filtrelenmesi platformun sürdürülebilirliği ve kullanıcı deneyimi için hayati öneme sahiptir. Bu proje otomatik bir sınıflandırma sistemi kurarak bu sürecin verimliliğini arttırmayı amaçlamaktadır.

1.2. Proje Hedefleri

- Apache Spark kullanarak metin verileri üzerinde bir makine öğrenimi boru hattı oluşturmak.
- Stack Overflow sorularının kalitesini tahmin etmek için sınıflandırma modeli geliştirmek.
- NVIDIA RAPIDS Accelerator for Apache Spark'ı entegre ederek boru hattının kritik adımlarında GPU hızlandırmanın uygulanabilirliğini ve performans kazançlarını iki farklı entegrasyon stratejisiyle değerlendirmek.
- CPU tabanlı referans uygulaması ile iki GPU hızlandırmalı uygulama arasında performans ve doğruluk karşılaştırması yapmak.
- Büyük veri işleme ve makine öğrenimi süreçlerindeki olası darboğazları analiz etmek.

2. Metodoloji

Proje; veri yükleme ve temizleme, özellik mühendisliği (metin tabanlı ve sayısal), model eğitimi ve tahmin adımlarını içeren bir yaklaşımla geliştirilmiştir. GPU hızlandırma, iki ana stratejiyle test edilmiş ve CPU tabanlı bir referans uygulaması ile karşılaştırılmıştır:

1. Spark'ın çekirdek operasyonları ve bazı Spark MLlib bileşenleri için RAPIDS Accelerator'ın otomatik entegrasyonu (401_sparkrapidsai.ipynb)
2. PySpark Pandas UDF'leri aracılığıyla cuML/cuDF kütüphanelerinin belirli özellik mühendisliği adımlarına manuel olarak entegrasyonu (401_sparkrapids_cuml.ipynb)
3. PySpark'ın standart CPU tabanlı operasyonlarını kullanarak bir referans boru hattı oluşturulmuştur (cpu.ipynb)

2.1. Veri Seti

Projede kullanılan veri seti Stack Overflow sorularından oluşan 60.000 adet kayıt içermektedir. Veri setinde Id, Title, Body, Tags, CreationDate ve Y sütunları bulunmaktadır. Y sütunu HQ (Yüksek Kaliteli), LQ_EDIT (Düşük Kaliteli Düzenlenmiş) ve LQ_CLOSE (Düşük Kaliteli Kapatılmış) olmak üzere üç dengeli sınıf içermektedir (her sınıftan 15.000 kayıt ile training tamamlanmıştır)

2.2. Veri Ön İşleme ve Temizleme

Veri ön işleme adımları, kirli veriyi temizlemek ve model için uygun formata dönüştürmek amacıyla Spark SQL/DataFrame API'leri kullanılarak gerçekleştirilmiştir.

- Title, Body, Tags, Y sütunlarındaki eksik değerler içeren satırlar kaldırılmıştır.
- Body sütunundaki HTML etiketleri (<.*?> regexi kullanılarak) regexp_replace fonksiyonu ile kaldırılmıştır.
- Title ve temizlenmiş Body sütunları concat_ws ile birleştirilerek tek bir text sütunu oluşturulmuştur.
- Tags sütunundaki etiketler (< ve > karakterleri kaldırılarak) boşluklara göre ayrılarak bir liste (tags_list) haline getirilmiştir.
- Y kalite etiketleri, when ve cast fonksiyonları kullanılarak sayısal label (HQ: 0.0, LQ_EDIT: 1.0, LQ_CLOSE: 2.0) değerlerine dönüştürülmüştür.

Bu adımların büyük çoğunluğu, özellikle Strateji 1'de RAPIDS Accelerator tarafından GPU'ya aktarılmış ve önemli hızlanmalar sağlamıştır. Strateji 2'de ise bazı UDF konfigürasyon eksiklikleri nedeniyle bu adımların bir kısmı CPU'ya düşmüştür.

2.3. Özellik Mühendisliği

Hem metinsel hem de sayısal özellikler çıkarılarak sınıflandırma modelinin performansı artırılmaya çalışılmıştır.

- **Skaler Özellikler:**
    1. title_len: Title sütununun karakter uzunluğu.
    2. body_len: CleanBody sütununun karakter uzunluğu.
    3. punct_count: text sütunundaki noktalama işaretlerinin (örneğin ?, !) sayısı.

4. avg_word_len: text sütunundaki ortalama kelime uzunluğu.
   Bu skaler özellikler Spark SQL/DataFrame API'leri (length, regexp_replace, size, split) kullanılarak hesaplanmış ve RAPIDS tarafından GPU'da hızlandırılmıştır.

- **Metinsel Özellikler:** Projede iki farklı metin özellik mühendisliği stratejisi uygulanmıştır:

  Strateji 1: Spark MLlib Bileşenleri ve RAPIDS Otomatik Hızlandırma (401_sparkrapidsai.ipynb)

  - text sütunu kelimelere (words) ayrılmıştır.
  - StopWordsRemover kullanılarak anlamsız kelimeler (filtered_words) çıkarılmıştır.
  - NGram ile 2'li kelime grupları (bigrams) oluşturulmuştur.
  - HashingTF ve IDF kullanılarak hem kelimeler (text_features) hem de 2-gramlar (bigrams_features) için vektörel temsiller oluşturulmuştur. Etiketler (tags_list) için de benzer şekilde HashingTF ve IDF (tags_tags) hesaplanmıştır.
  - Word2Vec modeli filtered_words üzerinden kelime gömme vektörleri (w2v_features) oluşturmak için eğitilmiştir.
  - Spark UI logları, bu pyspark.ml.feature adımlarının çoğunun (HashingTF, IDF, Word2Vec, VectorAssembler) org.apache.spark.ml.linalg.VectorUDT gibi Spark'ın kendi vektör tipleriyle çalışması ve RAPIDS'in bu spesifik Scala/Java UDF'leri için yerel GPU implementasyonlarına sahip olmaması nedeniyle CPU'ya düştüğünü göstermiştir. Bu aşamalarda GPU hızlandırma sınırlı kalmıştır. Ayrıca train_data.cache() ve test_data.cache() komutları, veriyi Disk Memory Deserialized seviyesinde CPU belleğinde tuttuğu için, önbelleğe alınan verilere erişildiğinde InMemoryTableScanExec operasyonları CPU'da kalmıştır.

  Strateji 2: Özel PySpark Pandas UDF'ler ve cuML/cuDF Kütüphaneleri (401_sparkrapids_cuml.ipynb)

  - Bu yaklaşım, pyspark.ml.feature kısıtlamalarını aşmak amacıyla geliştirilmiştir.
  - cuml.feature_extraction.text.TfidfVectorizer kullanılarak özel Pandas UDF'ler içinde cuDF ve cuML kütüphaneleri aracılığıyla doğrudan GPU üzerinde gerçekleştirilmesi amaçlanmıştır.
  - sparknlp.annotator.Word2VecApproach ayrı bir adım olarak CPU'da eğitilmiş, ardından bu gömmelerin belge bazında ortalaması bir Pandas UDF içinde cuPy kullanılarak GPU'da hesaplanması amaçlanmıştır.
  - Bu notebook'ta Pandas UDF'ler içinde Spark MLlib'in VectorUDT türlerinin kullanılması UDT'lerin Spark-RAPIDS tarafından doğrudan GPU'da columnar formatta işlenememesi nedeniyle CPU'ya veri transferine yol açmıştır. Bu durum StringIndexer gibi bazı Spark MLlib transformatörlerinde (object-based ifadeler içeren) ve vektörizasyon adımlarının Python UDF'leri içinde çalıştırılmasında gözlemlenmiştir. Ayrıca train_data.cache() ve test_data.cache() gibi cache() kullanımları, veriyi varsayılan olarak CPU belleğinde (Disk Memory Deserialized seviyesinde) tutar. Bu da önbelleğe alınmış verilere erişildiğinde InMemoryTableScanExec operasyonlarının

GPU'ya offload edilememesine ve CPU'da kalmasına neden olmuştur. GPU belleğinde etkili önbellekleme, özellikle Spark MLlib'deki VectorUDT sınırlamaları nedeniyle zorlayıcı olmuştur.

2.4. Model Seçimi ve Eğitimi

- Sınıflandırma görevi için LogisticRegression modeli seçilmiştir.
- GPU Hızlandırma için: Her iki stratejide de, model eğitimi aşamasında spark_rapids_ml.classification.LogisticRegression kullanılmıştır. Spark UI logları cuML entegrasyonu ile bu modelin eğitiminin GPU'da gerçekleştiğini doğrulamıştır.
- CPU Karşılaştırması için: pyspark.ml.classification.LogisticRegression'ın standart CPU versiyonu kullanılmıştır.

2.5. Model Değerlendirme

Model performansı, test veri seti üzerinde MulticlassClassificationEvaluator kullanılarak değerlendirilmiştir. Temel metrikler olarak Accuracy ve F1 Skoru kullanılmıştır. Ayrıca modelin sınıflandırma detaylarını göstermek için confusion matrix çıkarılmıştır.

3. Kurulum

Proje Ubuntu 22.04 üzerinde aşağıdaki donanım ve yazılım bileşenleri kullanılarak yürütülmüştür:

- **Donanım:**
  - CPU: 11th Gen Intel(R) Core(TM) i7-11370H
  - RAM: 16 GB
  - GPU: NVIDIA GeForce RTX 3050
- **Yazılım:**
  - Spark Sürümü: Apache Spark 3.4.2 (PySpark)
  - Python Sürümü: 3.10.x
  - RAPIDS Sürümü: 24.02 (cuDF, cuML, cuPy)
  - Spark NLP Sürümü: 5.3.3
  - Diğer Kütüphaneler: pandas, numpy, scikit-learn, matplotlib, seaborn

Spark uygulamaları local[*] master modunda çalıştırılmış ve GPU kullanımı için aşağıdaki temel Spark yapılandırmaları uygulanmıştır.

- spark.plugins=com.nvidia.spark.SQLPlugin
- spark.rapids.sql.enabled=true
- spark.rapids.memory.hostStorageFraction=0.8
- spark.rapids.memory.deviceStorageFraction=0.8
- spark.rapids.sql.explain=ALL
- spark.sql.inMemoryColumnarStorage.batchSerializer=com.nvidia.spark.rapids.shims.SparkShimServiceProvider
- Strateji 1'e özel: spark.rapids.sql.rowBasedUDF.enabled=true (Bu konfigürasyon Strateji 2'nin notebook'unda eksikti, bu da bazı Pandas UDF'lerinin CPU'ya düşmesine neden oldu.)
- spark.rapids.sql.concurrentGpuTasks=2
- spark.memory.offHeap.enabled=true
- spark.memory.offHeap.size=8g

- spark.rapids.memory.gpu.allocation.limit=0.9
- spark.rapids.host.shim.async=true
- spark.sql.session.timeZone=UTC
- spark.sql.exec.CollectLimitExec=true

4. Sonuçlar

Projede hem GPU hızlandırmanın entegre edildiği iki farklı boru hattı (kimi zaman CPU fallback'ler ile) hem de tamamen CPU tabanlı bir boru hattı karşılaştırılmıştır. Süre karşılaştırmaları ve performans metrikleri sunulmuştur.

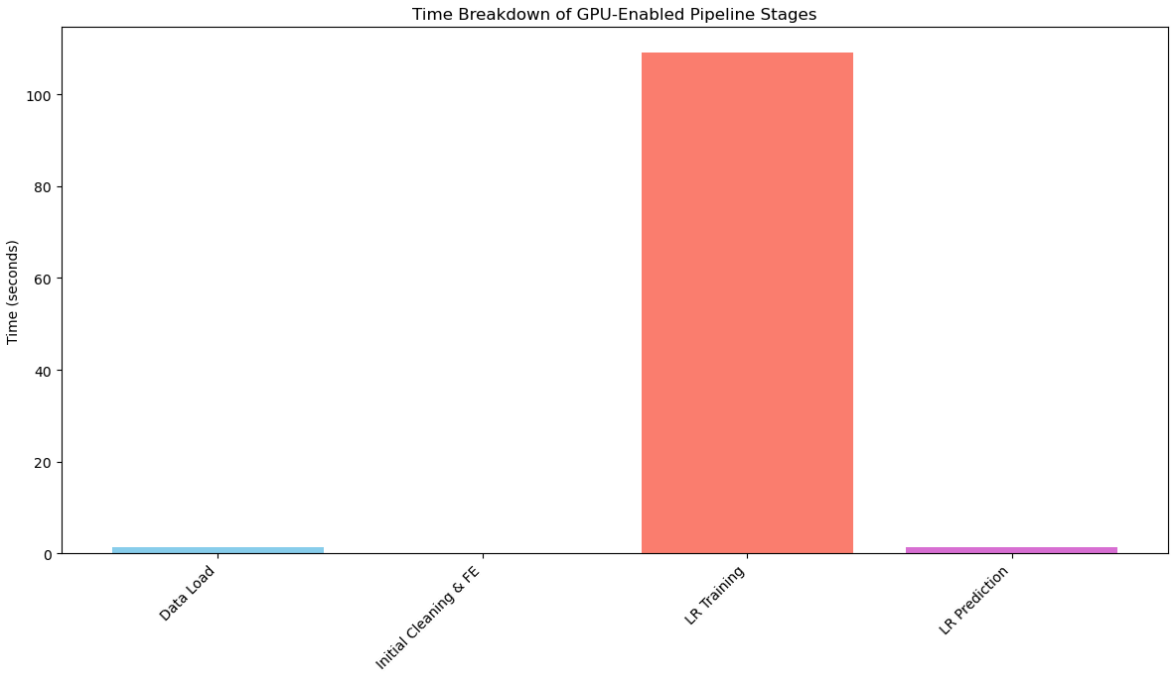
4.1. Süre Karşılaştırmaları

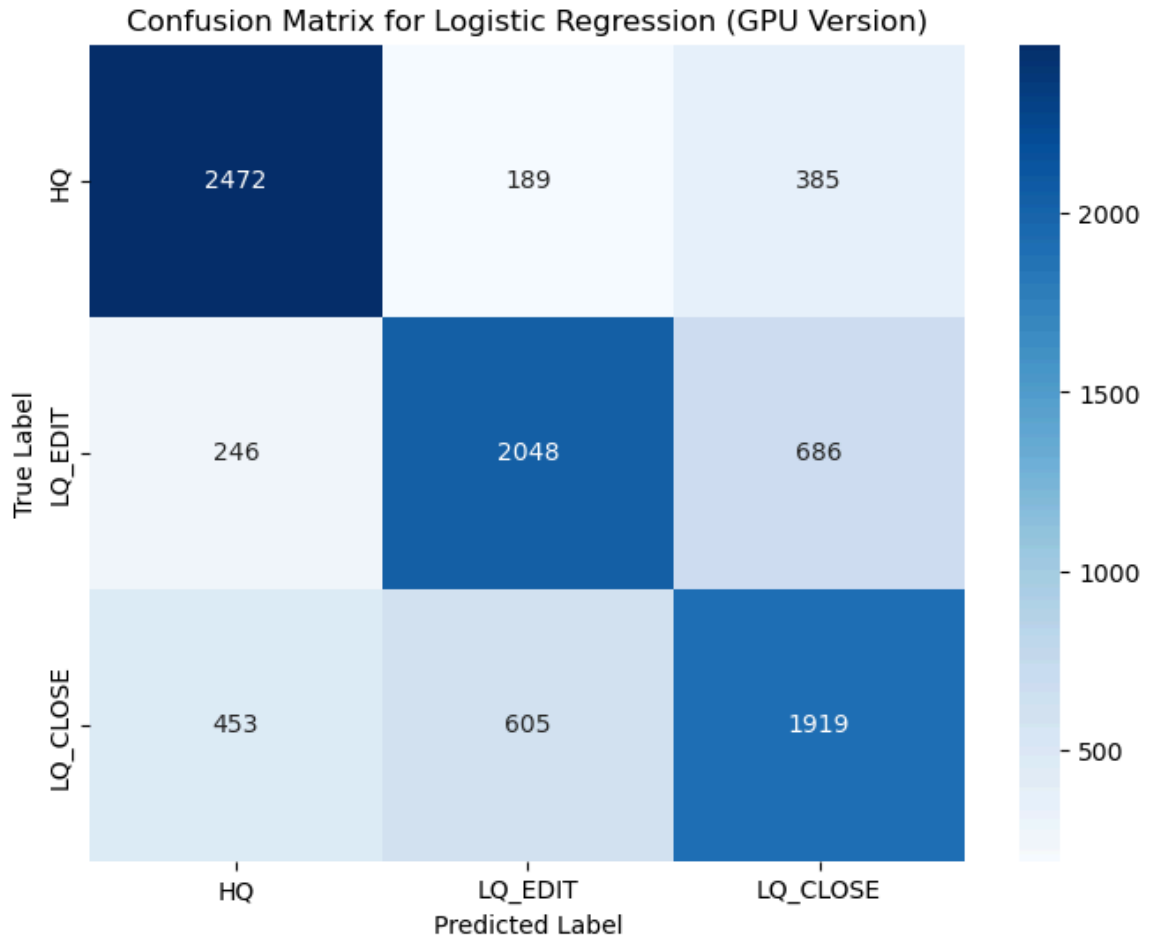
Aşama	Strateji 1 (Spark RAPIDS MLlib) Süresi (s)	Strateji 2 (PySpark Pandas UDF) Süresi (s)	CPU Boru Hattı Süresi (s)	Hızlanma Oranı (CPU / GPU) - Strateji 1	Hızlanma Oranı (CPU / GPU) - Strateji 2
Veri Yükleme	1.30	7.01	8.30	6.38x	1.18x
İlk Temizleme & Skaler FE	0.11	0.39	0.46	4.18x	1.18x
LR Eğitimi	109.19	147.53	119.54	1.09x	0.81x
LR Tahmini	1.44	0.76	0.46	0.32x	0.61x
Toplam Süre	**112.04**	**155.69**	**128.76**	**1.15x**	**0.83x**

LR Eğitimi ve Tahmini adımları boru hattının bütünsel bir parçası olduğundan ayrı bir süre olarak ölçülmüştür. Toplam süre; veri yükleme, ilk temizleme/skaler FE, LR Eğitimi ve LR Tahmini adımlarının toplamıdır.

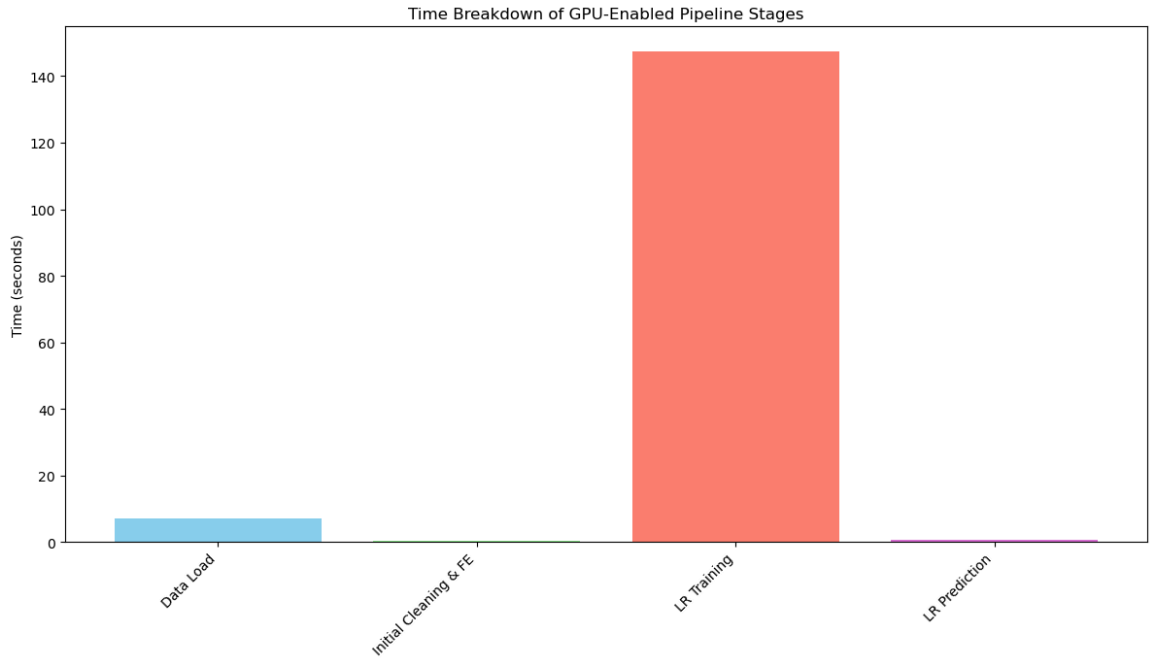
4.2. Model Performansı

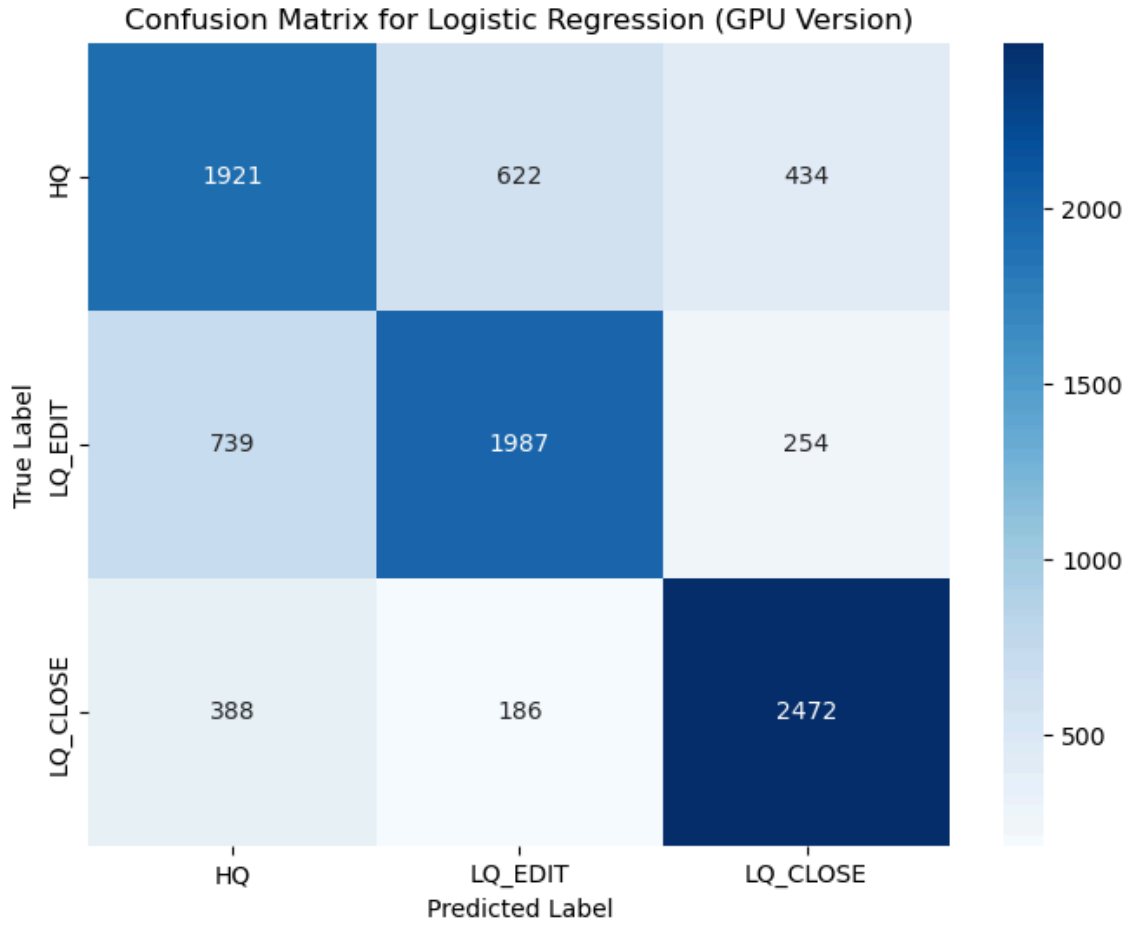
Strateji 1 (Spark RAPIDS MLlib) Performansı: Doğruluk: 0.7152 **F1 Skoru:** 0.7146



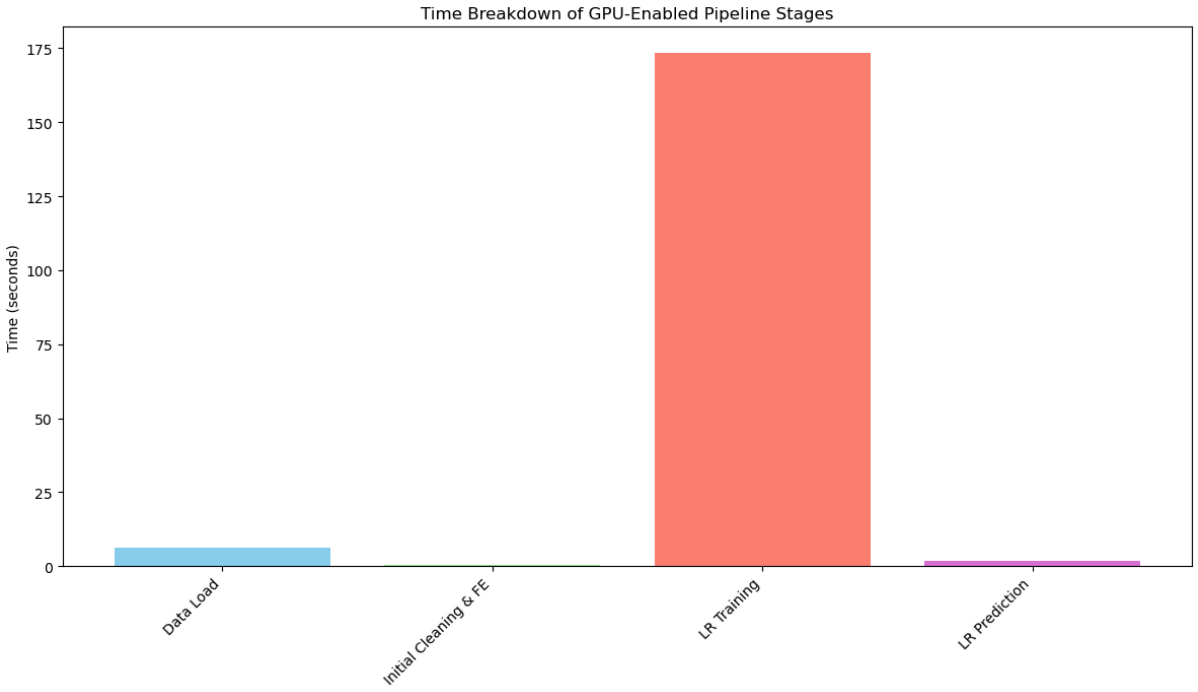


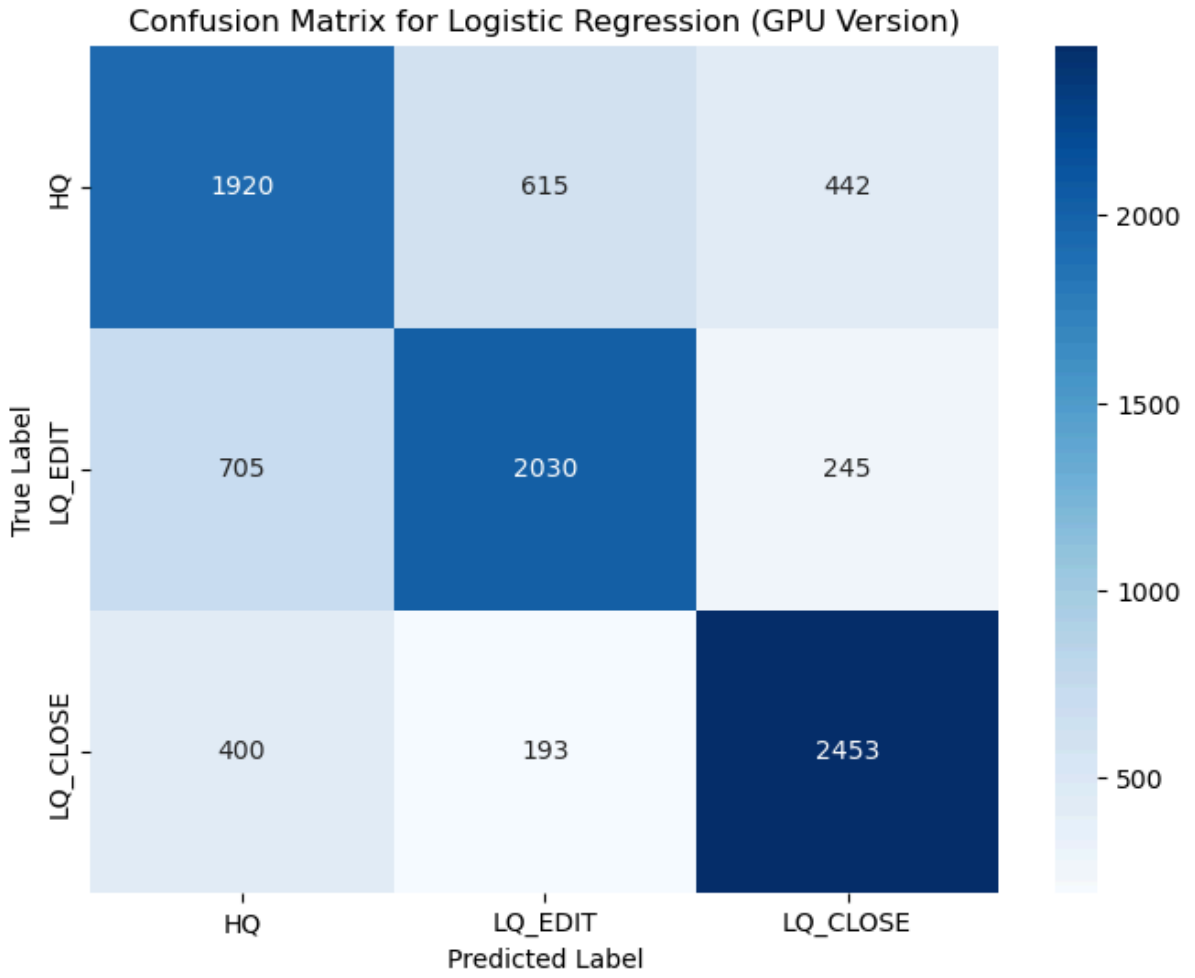
Strateji 2 (PySpark Pandas UDF ile cuML/cuDF) Performansı: Doğruluk: 0.7087 **F1 Skoru:** 0.7082



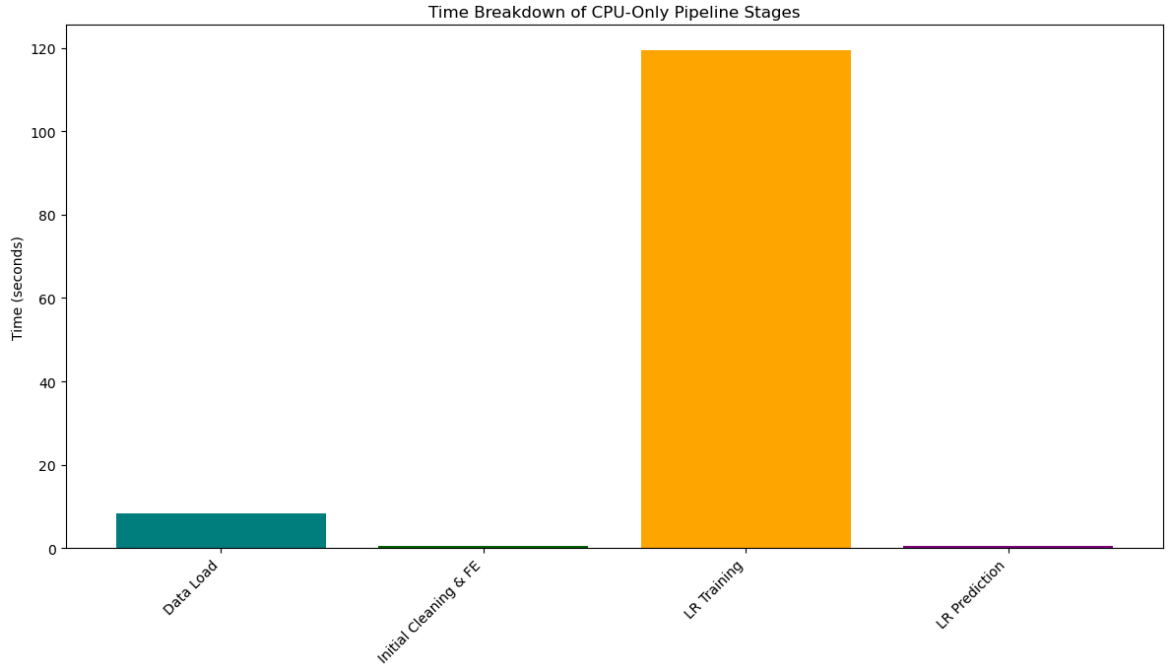


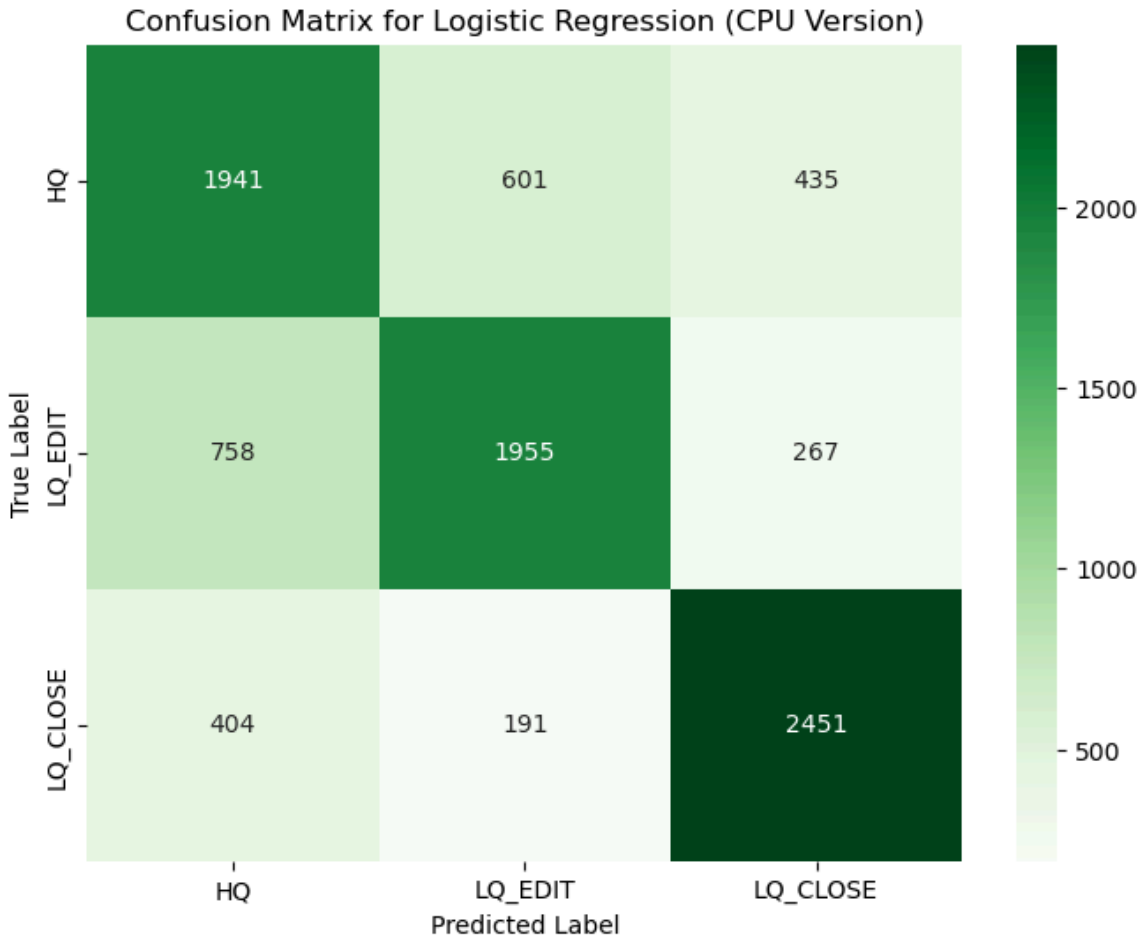
PySpark Pandas UDF ile cuML/cuDF için tekrar deneyiş:Accuracy: 0.7112, F1 Score: 0.7109





CPU Boru Hattı Performansı: Doğruluk: 0.7050 **F1 Skoru:** 0.7047





5. Tartışma

- **GPU Hızlandırmanın Etkinliği:**
    - Veri yükleme ve temel veri temizliği (HTML kaldırma, metin birleştirme, uzunluk hesaplamaları) gibi Spark SQL operasyonları NVIDIA RAPIDS Accelerator sayesinde önemli ölçüde GPU'da hızlandırılmıştır. Özellikle veri yüklemede (CPU'ya göre Strateji 1 için ~6.38x, Strateji 2 için ~1.18x) ve ilk temizleme/skaler özellik mühendisliğinde (Strateji 1 için ~4.18x, Strateji 2 için ~1.18x) kayda değer hızlanmalar gözlemlenmiştir. Strateji 1'de, LR eğitiminde CPU'ya göre hafif bir hızlanma (~1.09x) elde edilmiştir.
    - Strateji 1 için: pyspark.ml.feature kütüphanesindeki StopWordsRemover, HashingTF, IDF, NGram, Word2Vec, VectorAssembler gibi transformatörler, Spark'ın dahili VectorUDT kullanımı ve RAPIDS'in bu spesifik Scala/Java UDF'leri için henüz tam GPU desteği sunmaması nedeniyle yoğun bir şekilde CPU'ya düşmüştür. Spark logları bu adımların GPU'da çalışamadığını ve verinin CPU ile GPU arasında sürekli dönüştürülmesi gerektiğini belirtmiştir. Bu durum boru hattının özellik mühendisliği aşamasında bir darboğaz yaratmıştır.
    - Strateji 2 için: pyspark.ml.feature'daki kısıtlamaların aşılması hedeflense de uygulandığı haliyle beklenen performansa tam olarak ulaşılamamıştır. train_data.cache() ve test_data.cache() komutları, veriyi Disk Memory Deserialized seviyesinde CPU belleğinde tutmuştur. Bu da, önbelleğe alınan verilere erişildiğinde GPU'ya offload edilememesine ve InMemoryTableScanExec gibi operasyonların CPU'da kalmasına neden olmuştur. GPU belleğinde etkin önbellekleme MLlib'deki VectorUDT sınırlamaları nedeniyle zorlayıcı olmuştur.
    - Strateji 1 (Spark RAPIDS MLlib) ile elde edilen yaklaşık %71.5'lik doğruluk ve F1 skoru, CPU-Only boru hattının (%70.5 doğruluk) performansından biraz daha iyi sonuç vermiştir.
    - Strateji 2'deki doğruluk düşüşü (%70.8) ve genel performans (toplam süre CPU'dan daha yavaş: ~0.83x hızlanma oranı), büyük ölçüde CPU fallback'leri ve TfidfVectorizer'ın her batch'te yeniden eğitilmesi gibi sorunlardan kaynaklanmıştır. LR Tahmin adımı, her iki GPU stratejisinde de CPU'dan daha yavaş gerçekleşmiştir (Strateji 1 için ~0.32x hızlanma oranı, Strateji 2 için ~0.61x hızlanma oranı).

6. Sonuç

Bu proje Stack Overflow soru kalitesi tahmini için Apache Spark üzerinde GPU hızlandırmalı bir makine öğrenimi boru hattı geliştirme sürecini tamamlamıştır. NVIDIA RAPIDS Accelerator for Apache Spark'ın veri yükleme ve temel DataFrame operasyonlarında sağladığı önemli performans kazançları gösterilmiştir. Bununla birlikte pyspark.ml.feature kütüphanesindeki mevcut sınırlamalar ve özellikle Strateji 2'de gözlemlenen yanlış konfigürasyonlar nedeniyle özellik mühendisliği aşamalarında CPU darboğazları yaşandığı ve bu durumun genel boru hattı hızlanmasını ve model doğruluğunu etkilediği tespit edilmiştir.

Strateji 1 genel olarak CPU'ya kıyasla hafif bir hızlanma ve biraz daha iyi doğruluk sunmuştur. Strateji 2 ise VectorUDT kullanımı ve UDF'lerdeki veri transferi maliyetleri nedeniyle beklenen performansın gerisinde kalmış ve genel olarak CPU'dan daha yavaş çalışmıştır. Özel Pandas UDF'ler aracılığıyla cuML/cuDF entegrasyonu, bu kısıtlamaları aşmak için umut verici bir alternatif sunsa da, doğru konfigürasyon ve uygulama stratejisinin kritik önemi de anlaşılmıştır. Elde edilen %70 civarı doğruluk oranı, gelecekteki iyileştirmeler için potansiyel barındırmaktadır.

7. Yapılabilecek Çalışmalar

- Logistic regression modelinin ve özellik mühendisliği bileşenlerinin (numFeatures:HashingTF, vectorSize:Word2Vec, maxIter:LogisticRegression) hiperparametrelerinin cross validation ile optimize edilmesi
- Spark NLP üzerinden önceden eğitilmiş GloVe veya FastText gibi daha zengin kelime gömme modellerinin kullanımı
- spark_rapids_ml veya XGBoost gibi GPU hızlandırmalı diğer sınıflandırma modellerinin performansının test edilmesi
- RAPIDS tarafından henüz GPU desteği verilmeyen pyspark.ml.feature bileşenleri için özel cuML/cuDF entegrasyonlarının daha kapsamlı bir şekilde geliştirilmesi ve test edilmesi