Домашна 3 – Бојана Андонова 221225

Оваа домашна се базира на модерна архитектура која ги комбинира **Apache Spark** за дистрибуирана обработка на големи податоци и **Apache Kafka** за стриминг на податоци. Проектот го поделив на две логички целини: **офлајн фаза**, каде што се врши анализа, трансформација на податоците и тренирање на моделите и **онлајн фаза**, каде што најдобриот модел се користи за давање предвидувања за нови пациенти во реално време.

Подготовка на Податоците

Првиот чекор беше правилно да ги поделам податоците со цел да симулирам реална ситуација каде што имаме историски податоци за тренинг и нови податоци кои пристигнуваат. Креирав скрипта scripts/split_data.py која го дели оригиналниот сет на два дела: offline.csv (80% од податоците) и online.csv (20% од податоците).

Значајно е да напоменам дека користев **стратификирана поделба (stratified split)** врз основа на целната колона Diabetes_binary. Ова беше клучно бидејќи сакав соодносот на позитивни и негативни класи (луѓе со и без дијабетес) да остане ист и во тренирачкото множество и во множеството за тестирање. Доколку користев обична поделба, постоеше ризик едното множество да биде нерепрезентативно, што би довело до погрешна евалуација на моделите.

Офлајн Фаза

Во офлајн фазата, со цел да изградам модел кој со најголема точност ќе може да го предвиди дијабетесот, ја користев библиотеката **PySpark Mllib**.

Пред да започнам со тренирањето, податоците мораа да бидат соодветно подготвени. Во скриптата scripts/transformations.py дефинирав pipeline кој се состои од два главни чекори:

1. **VectorAssembler**: Бидејќи Spark ML алгоритмите очекуваат влезните карактеристики да бидат во форма на еден вектор, ги споив сите 21 колони (како HighBP, BMI, Age итн.) во една векторска колона наречена features.

2. **StandardScaler**: Ова е многу важен чекор, особено за алгоритми како Логистичка Регресија. Бидејќи атрибутите имаат различни рангови (на пример, BMI може да биде 30, додека HeartDiseaseorAttack е 0 или 1), без скалирање, атрибутите со поголеми вредности би доминирале во моделот. Со StandardScaler ги нормализирав податоците така што имаат стандардна девијација, ставајќи ги сите атрибути на иста скала.

Одлучив да експериментирам со три различни класификациски алгоритми за да видам кој најдобро се однесува на овој проблем:

1. **Logistic Regression**: Едноставен, но моќен алгоритам за бинарна класификација. Експериментирав со параметрите за регуларизација (regParam) и ElasticNet мешање (elasticNetParam).

2. **Random Forest**: Метод кој креира повеќе дрва на одлука. Тука ги варирав бројот на дрва (numTrees) и максималната длабочина (maxDepth).

3. **Decision Tree**: Основен модел кој е лесен за интерпретација. Ги оптимизирав длабочината на дрвото и минималниот број на инстанци во јазол.

За секој од овие модели користев **Cross-Validation** со 3-folds. Ова значи дека Spark автоматски го делеше тренинг сетот на 3 дела, трениираше на 2 и тестираше на 1, со цел да се добие пореален резултат за перформансите.

Резултатите од тренирањето покажаа дека сите модели даваат солидни и приближно слични резултати.

```
LogisticRegression        | F1: 0.8303 | Time: 18.67s
RandomForest              | F1: 0.7962 | Time: 18.35s
DecisionTree              | F1: 0.8272 | Time: 9.79s
```

```
==================================
BEST MODEL: LogisticRegression
BEST F1 SCORE: 0.8303
==================================
```

На сликата погоре се прикажани резултатите од извршувањето на train_model.py. Во табелата јасно се гледа споредбата на F1 резултатите и времето потребно за тренирање за секој алгоритам.

Според резултатите, **Logistic Regression** покажа највисок F1 резултат од приближно **0.8303**. Поради тоа, овој модел беше автоматски зачуван како best_diabetes_model и искористен во следната фаза.

Онлајн Фаза

Откако го имав најдобриот модел, следниот предизвик беше да го ставам во функција на систем кој работи во реално време.

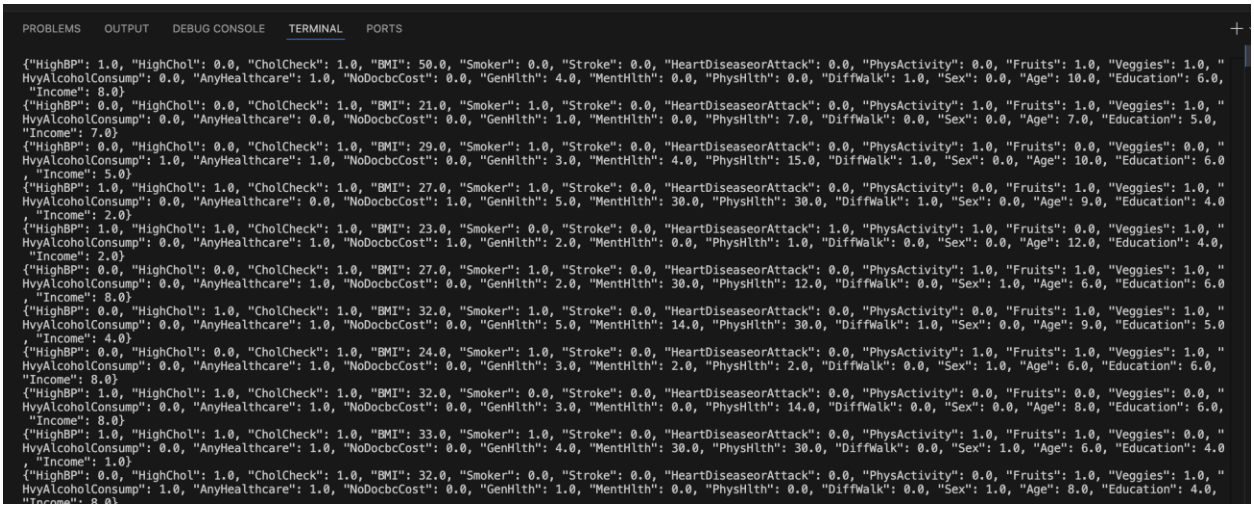
За реализација на онлајн фазата, ја искористив претходната домашна каде имаше инфраструктура во која беа дефинирани Docker контејнери за:

- **Zookeeper**: Сервис за координација потребен за Kafka.

- **Kafka Broker**: Централниот дел на системот кој овозможува испраќање и примање на пораки (events) помеѓу различни компоненти.

За да симулирам пристигнување на нови пациенти во болница, напишав Python скрипта producer.py, исто така слична како во претходната домашна задача. Оваа скрипта:

1. Ги чита податоците ред по ред од CSV датотеката.
2. Ја отстранува колоната Diabetes_binary, бидејќи во реалноста не знаеме дали новиот пациент има дијабетес - тоа е она што сакаме да го предвидиме.
3. Ги пакува податоците во **JSON формат**.
4. Ги испраќа пораките кон Kafka topic-от health_data со мало доцнење од 50 милисекунди, за да се добие ефект на стриминг.



Сликата го прикажува терминалот каде што работи Producer-от. Може да се забележи како податоците за секој пациент се испраќаат во JSON формат кон Kafka кластерот во реално време.

Последната и најважна компонента е inference.py. Ова е апликација напишана во Spark Structured Streaming која работи континуирано.

Процесот на инференција се одвива во неколку чекори:

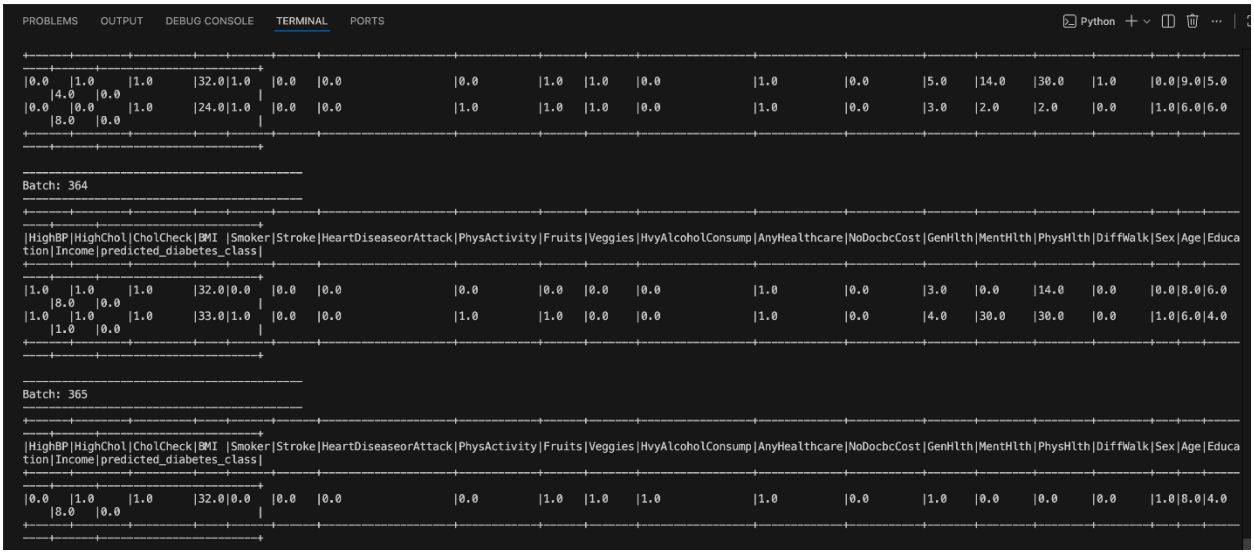
1. Апликацијата на почетокот го вчитува претходно зачуваниот best_diabetes_model.
2. Spark се поврзува со Kafka topic-от health_data и ги презема новите пораки штом пристигнат.
3. JSON пораките се декодираат и се претвораат во Spark DataFrame со дефинирана шема.

4. Врз парсираните податоци се применува вчитаниот модел. Моделот додава нова колона prediction (во кодот алијасирана како predicted_diabetes_class) која содржи 1.0 (дијабетес) или 0.0 (нема дијабетес).

5. Резултатите се испраќаат на две места:

    - Назад во Kafka на нова тема health_data_predicted

    - На конзола за да можеме визуелно да ги потврдиме резултатите.



На оваа слика е прикажан излезот од Inference скриптата. Се гледа табеларниот приказ на стриминг податоците каде што покрај влезните атрибути, за секој пациент е генерирана и предвидената класа во последната колона.

Заклучок

Со реализацијата на овој проект, успешно демонстрирав како техниките на рударење на податоци можат да се применат во комплексен, дистрибуиран систем. Клучните придобивки од овој систем се:

- **Автоматизација**: Целиот процес од примање на податоци до предвидување е автоматизиран.

- **Скалабилност**: Благодарение на Spark и Kafka, овој систем може лесно да се скалира за да обработува милиони записи во секунда, што е невозможно со традиционални скрипти.

- **Модуларност**: Офлајн и онлајн деловите се независни. Можеме да тренираме покомплексни модели офлајн и само да го замениме фајлот со моделот во онлајн делот без да го прекинуваме стримингот на податоци.