eBook

# 데이터 사이언스 사용 사례 Big Book

코드 샘플, 노트북을 비롯한
기술 블로그 컬렉션

databricks

# 목차

1장:
소개 3

2장:
Databricks로 금융 시계열 분석 민주화 4

3장:
동적 시간 왜곡 및 MLflow를 사용한 매출 동향 발견 시리즈
**1부:** 동적 시간 왜곡의 이해 14
**2부:** 동적 시간 왜곡 및 MLflow를 사용한 매출 동향 발견 20

4장:
안전 재고 분석에 대한 새로운 접근 방식으로 재고를 최적화하는 방법 28

5장:
공급망 수요 예측을 개선하는 새로운 방법 34

6장:
Prophet 및 Apache Spark를 사용한 세분화된 대규모 시계열 예측 44

7장:
Databricks에서 결정 트리 및 MLflow로 대규모 금융 사기 탐지 51

8장:
Virgin Hyperloop One에서 Koalas로 처리 시간을 몇 시간에서 몇 분으로 단축한 비결 62

9장:
Databricks에서 Apache Spark를 사용하여 개인화된 쇼핑 환경 제공 70

10장:
Databricks에서 Apache SparkR로 대규모 시뮬레이션 병렬화 75

11장:
고객 사례 분석 78

1장:

# 소개

데이터 사이언스의 세계는 발전 속도가 너무나 빨라서 현재 개발하려는 주제와 관련된 실제 사용 사례를 찾기가 어렵습니다. 그래서 산업 분야의 선구적 이론가들이 운영하는 블로그에서 지금 바로 사용할 수 있는 실용적 사용 사례를 모았습니다. 이 입문 참조 가이드는 Databricks 플랫폼을 사용해볼 수 있도록 코드 샘플을 포함하여 모든 필수 정보를 제공합니다.

2장:

# Databricks로 금융 시계열 분석 민주화

## Databricks Connect 및 Koalas로 개발 속도 단축

글: **Ricardo Portilla**

2019년 10월 09일

Databricks에서 이 노트북을 확인해보세요 →

## 소개

금융 기관에서 데이터 사이언티스트, 데이터 엔지니어, 분석 전문가는 수천억 달러에 달하는 자산을 보호하고, 투자자를 가령 주식 시장 폭락과 같은 *엄청난 재정적 피해*로부터 보호하는 등의 역할을 합니다. 이런 문제의 기저에 있는 가장 큰 기술적 어려움은 시계열 조작을 확장하는 것입니다. 금융 기관에서 사용하는 리치 데이터 소스로는 틱 데이터, 대체 데이터(예: 위치 정보 데이터, 거래 데이터), 기본 경제 데이터 등이 있으며, 이들 모두 기본적으로 타임스탬프를 기준으로 색인됩니다. 위험, 사기, 규정 준수 등과 같은 금융 비즈니스 문제를 해결하려면 궁극적으로는 수천 개의 시계열을 집계하고 동시에 분석하는 능력이 있어야 합니다. RDBMS 기반의 기존 기술은 거래 전략을 분석하거나, 과거 데이터에 대한 규정 분석을 실시할 때 수년 치에 대한 데이터로 확장하기가 쉽지 않습니다. 게다가 대부분 기존 시계열 기술은 표준 SQL이나 Python 기반 API가 아니라 전문 언어를 사용합니다.

다행히 Apache Spark™에는 시계열 작업을 자동으로 병렬화하는 윈도잉 등의 기본 기능이 여러 가지 내장되어 있습니다. 그뿐만 아니라 친숙한 pandas 구문을 사용하여 Apache Spark에서 분산된 머신 러닝 쿼리를 사용할 수 있는 오픈 소스 프로젝트인 Koalas를 사용하면 데이터 사이언티스트와 분석 전문가까지 이 기능을 이용할 수 있습니다.

이 블로그에서는 수만 개의 티커에 대한 시계열 함수를 구축하는 방법을 알아보겠습니다. 그다음에는 Databricks Connect로 로컬 IDE에서 함수를 모듈화하고, 리치 시계열 특징 집합을 생성하는 방법을 살펴봅니다. 마지막으로 금융 분야의 이상치 탐지나 다른 통계 분석에 사용할 데이터 준비를 확장하고자 하는 pandas 사용자를 위해, 시장 조작 예시를 통해 Koalas로 일반적 데이터 사이언스 워크플로에 투명하게 확장하는 방법을 보여드릴 것입니다.

databricks

## 시계열 데이터 소스 설정

먼저 몇 가지 기존 금융 시계열 데이터 세트(거래, 호가)를 입력하는 것으로 시작해 봅시다. 이 블로그에 사용할 데이터 세트는 거래소와 National Best Bid Offer(NBBO) 피드(NYSE 등의 거래소)에서 받은 데이터를 기반으로 시뮬레이션한 데이터입니다. 예시 데이터는 다음 링크에서 확인할 수 있습니다. **example data here**.

이 블로그에서는 대체로 기본적 금융 용어를 사용합니다. 더욱 광범위한 참고 자료는 Investopedia **문서**를 참조하세요. 아래의 데이터 세트는 각 타임스탬프에 대해 `TimestampType`을 할당했습니다. 거래 실행 시간과 호가 변경 시간은 정규화를 위해 `event_ts`로 변경했습니다. 또한, 이 블로그에 첨부된 노트북을 확인해보면 아시겠지만 최종적으로는 데이터 세트를 Delta 형식으로 변환합니다. 이는 **데이터 품질**을 보장하고 아래와 같은 인터랙티브 쿼리 타입에 가장 효율적인 열 형식을 유지하기 위해서입니다.

```
trade_schema = StructType([
     StructField(“symbol”, StringType()),
     StructField(“event_ts”, TimestampType()),
     StructField(“trade_dt”, StringType()),
     StructField(“trade_pr”, DoubleType())
 ])

 quote_schema = StructType([
     StructField(“symbol”, StringType()),
     StructField(“event_ts”, TimestampType()),
     StructField(“trade_dt”, StringType()),
     StructField(“bid_pr”, DoubleType()),
     StructField(“ask_pr”, DoubleType())
 ])
```

```
display(spark.read.format("delta").load("/tmp/finserv/delta/trades"))
```

(1) Spark Jobs

| symbol | event_ts | trade_dt | trade_pr |
|---|---|---|---|
| AMH | 2017-08-31T11:58:35.000+0000 | 2017-08-31 | 347.3411812850558 |
| EMIS | 2017-08-31T22:52:54.000+0000 | 2017-08-31 | 348.2907055152273 |
| AMH | 2017-08-31T04:33:52.000+0000 | 2017-08-31 | 346.3701388789535 |
| AMH | 2017-08-31T02:32:37.000+0000 | 2017-08-31 | 346.3012590012465 |
| KIO | 2017-08-31T06:03:36.000+0000 | 2017-08-31 | 349.5138613212247 |
| EMIS | 2017-08-31T18:00:38.000+0000 | 2017-08-31 | 348.0215275764011 |
| EMIS | 2017-08-31T03:39:54.000+0000 | 2017-08-31 | 348.5171330367943 |
| EMIS | 2017-08-31T02:59:52.000+0000 | 2017-08-31 | 348.54131225455575 |
| KWR | 2017-08-31T10:02:30.000+0000 | 2017-08-31 | 348.86337472824437 |

Showing the first 1000 rows.

```
display(spark.read.format("delta").load("/tmp/finserv/delta/quotes"))
```

(1) Spark Jobs

| symbol | event_ts | trade_dt | bid_pr | ask_pr |
|---|---|---|---|---|
| COST | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 343.69295468812896 | 350.909849275807 |
| AMD | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 347.04709899077204 | 348.5183895843159 |
| KYN | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 348.53269061203054 | 351.4189643371137 |
| KYN | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 344.7049081216955 | 349.80283794725966 |
| KYN | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 346.216800782748 | 348.6772930682145 |
| EMIS | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 349.4801250232342 | 351.0930879023341 |
| EMIS | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 346.94067005458623 | 348.7309464067882 |
| EMIS | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 346.54222291125706 | 348.25466426470564 |
| CAF | 2017-08-31T00:10:19.000+0000 | 2017-08-31 | 348.11208695271176 | 352.34177898766933 |

Showing the first 1000 rows.

## Apache Spark로 시계열 병합 및 집계

현재 전 세계 금융 시장에는 공개 거래되는 증권의 개수는 60만 개 이상입니다. 우리가 가진 거래 및 호가 데이터 세트도 이 정도 규모의 증권이 기반하므로 쉽게 확장시킬 수 있는 도구가 필요합니다. Apache Spark는 ETL에 간단한 API를 제공하고, 병렬화를 위한 표준 엔진이므로 표준 지표를 병합하고 집계하는 도구로 사용하겠습니다. 우리가 유동성, 위험, 사기를 이해하는 데 도움이 될 것입니다. 먼저 거래와 호가를 병합한 다음, 거래 데이터 세트를 집계하여 데이터를 간단히 슬라이스하는 방법을 보여드리겠습니다. 마지막으로 이 코드를 클래스로 패키징하여 Databricks Connect에서 반복적 개발 시간을 단축하는 방법을 보여드릴 것입니다. 다음 페이지의 지표에 사용한 전체 코드는 노트북에 첨부했습니다.

**조인 기준(As-of join)**

조인 기준(as-of join)은 일반적으로 사용하는 '병합' 기술로, 좌측 타임스탬프에서 유효한 가장 최근의 우측 값을 반환합니다. 대부분 시계열 분석의 경우, 여러 시계열 유형을 심볼에 결합하여 다른 시계열(예: 거래)에 있는 특정 시점에서 어느 시계열(예: NBBO)의 상태를 파악합니다. 아래의 예시에서는 모든 심볼에서 각 거래에 대한 NBBO의 상태를 기록합니다. 아래의 그림에서 볼 수 있듯이, 초기 기본 시계열(거래)에서 시작해서 NBBO 데이터 세트를 병합하였고 각 타임스탬프에는 가장 최근 입찰과 매수가 "거래 시점 기준"으로 기록되어 있습니다. 가장 최근 입찰과 매수를 알면 그 차이(스프레드)를 계산하여 어느 시점에 유동성이 낮아졌는지 파악할 수 있습니다(스프레드가 큰 경우). 이런 지표는 *알파*를 높이기 위한 거래 전략을 개선하는 데 도움이 됩니다.

먼저 시간순으로 정렬한 다음, 기본 windowing 함수를 사용해서 **마지막** null이 아닌 호가 값을 찾습니다.

```
# sample code inside join method

#define partitioning keys for window
partition_spec = Window.partitionBy('symbol')

# define sort - the ind_cd is a sort key (quotes before trades)
join_spec = partition_spec.orderBy('event_ts'). \
            rowsBetween(Window.unboundedPreceding, Window.currentRow)

# use the last_value functionality to get the latest effective record
select(last("bid", True).over(join_spec).alias("latest_bid"))
```

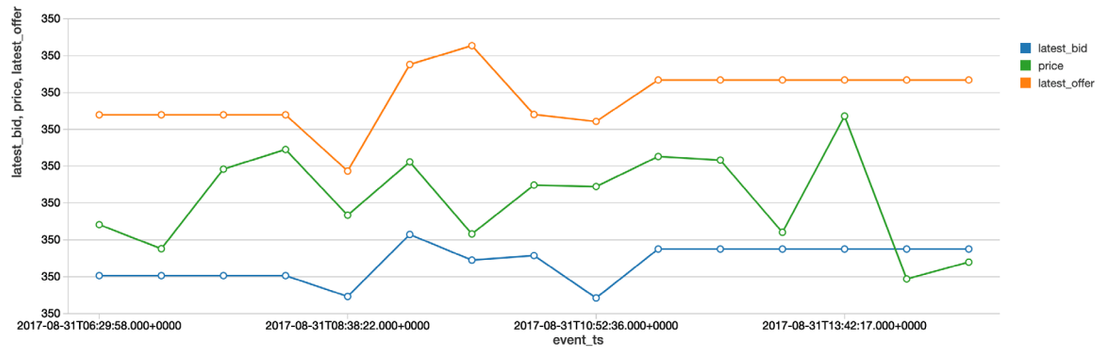
이제 사용자 정의 조인을 호출해서 데이터를 병합하고 호가를 연결합니다. 전체 코드는 첨부한 노트북을 참조하세요.

```
# apply our custom join
mkt_hrs_trades = trades.filter(col("symbol") == "K")
mkt_hrs_trades_ts = base_ts(mkt_hrs_trades)
quotes_ts = quotes.filter(col("symbol") == "K")

display(mkt_hrs_trades_ts.join(quotes_ts))
```

```
1 display(mkt_hrs_trades_ts.join(quotes_ts))
```

▸ (5) Spark Jobs

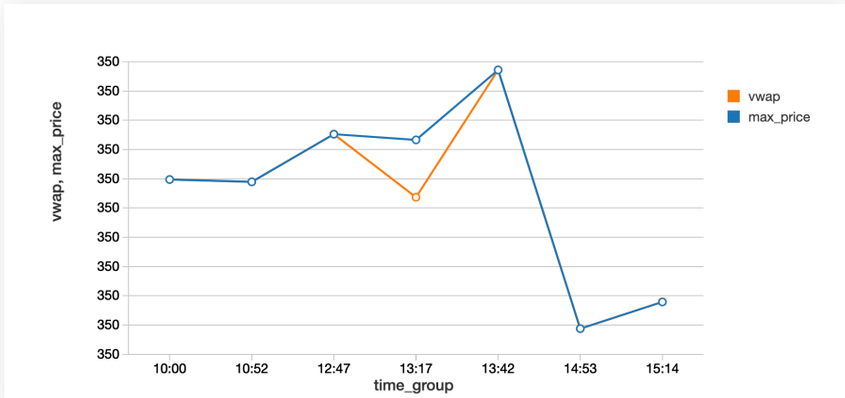
| event_ts | price | symbol | ind_cd | latest_bid | latest_offer |
|---|---|---|---|---|---|
| 2017-08-31T06:29:58.000+0000 | 347.4121586706382 | K | 1 | 346.0297772384752 | 350.39315623662594 |
| 2017-08-31T06:32:30.000+0000 | 346.7582132240916 | K | 1 | 346.0297772384752 | 350.39315623662594 |
| 2017-08-31T06:37:31.000+0000 | 348.919146315238 | K | 1 | 346.0297772384752 | 350.39315623662594 |
| 2017-08-31T06:56:24.000+0000 | 349.45235333868743 | K | 1 | 346.0297772384752 | 350.39315623662594 |
| 2017-08-31T08:38:22.000+0000 | 347.6687817715506 | K | 1 | 345.46234681384203 | 348.8679460367136 |
| 2017-08-31T08:52:59.000+0000 | 349.11648025163987 | K | 1 | 347.1462324487709 | 351.7600135730971 |
| 2017-08-31T09:22:55.000+0000 | 347.16036576622395 | K | 1 | 346.44863456258196 | 352.27114879065283 |
| 2017-08-31T10:00:54.000+0000 | 348.4869310969907 | K | 1 | 346.5728444681869 | 350.40101772579453 |
| 2017-08-31T10:52:36.000+0000 | 348.44707325529976 | K | 1 | 345.42173015910055 | 350.21440300934785 |




**거래 패턴에 대한 VWAP 표시**

앞서 병합 기술에 대해 살펴보았으므로 이제 표준 집계, 즉 거래량 가중 평균가(VWAP: Volume-Weighted Average Price)에 초점을 맞추겠습니다. 이 지표는 하루 중 증권 흐름과 가격을 나타냅니다. (첨부된 노트북에 있는) wrapper 클래스의 VWAP 함수는 VWAP가 증권 거래 가격의 위에 위치하는지, 아래에 위치하는지 보여줍니다. 특히, VWAP(주황색)가 거래 가격 아래에 있는 기간을 찾으면 해당 증권이 과매수되었다는 것을 나타낼 수 있습니다.

```
trade_ts = base_ts(trades.select('event_ts', symbol, 'price', lit(100).
alias("volume")))
vwap_df = trade_ts.vwap(frequency = 'm')

display(vwap_df.filter(col(symbol) == "K") \
    .filter(col('time_group').between('09:30','16:00')) \
    .orderBy('time_group'))
```

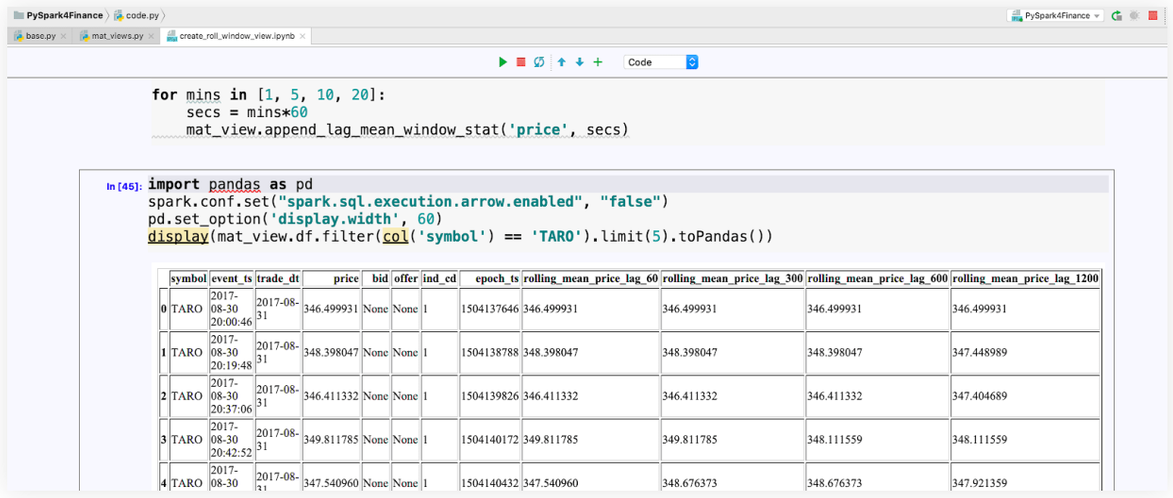


### Databricks Connect를 사용한 빠른 반복적 개발

지금까지 일회성 시계열 지표에 대한 기본 래퍼를 만들었습니다. 그러나 코드를 프로덕션화하려면 모듈화와 테스트가 필요하고, 이는 IDE에서 작업하는 것이 가장 좋습니다. 올해 출시된 Databricks Connect는 로컬 IDE 개발 기능을 제공하고, 라이브 Databricks 클러스터에 대해 테스트하는 환경을 향상했습니다. 금융 분석에서 Databricks Connect를 사용할 경우, 소규모 테스트 데이터에 시계열 feature들을 추가할 수 있고 몇 년분의 과거 틱 데이터에 인터랙티브 Spark 쿼리를 유연하게 실행하여 feature들을 검증할 수 있다는 장점이 있습니다.

우리는 PyCharm을 사용하여 PySpark 함수를 래핑하는 데 필요한 클래스를 구성하고, 리치 시계열 feature set을 생성합니다. 이 IDE에서는 코드 완성, 서식 지정 표준, 코드 실행 전에 클래스와 메서드를 신속히 테스트할 수 있는 환경을 제공합니다.


클래스를 빠르게 디버깅한 다음, 노트북에서 직접 Spark 코드를 실행할 수 있습니다. 이때, Jupyter 노트북을 사용해서 로컬 클래스를 로드하고 확장할 수 있는 인프라로 인터랙티브 쿼리를 실행합니다. 콘솔 창에는 라이브 클러스터에 대해 실행되는 작업이 표시됩니다.



```
for mins in [1, 5, 10, 20]:
    secs = mins*60
    mat_view.append_lag_mean_window_stat('price', secs)
```

```
In [45]: import pandas as pd
         spark.conf.set("spark.sql.execution.arrow.enabled", "false")
         pd.set_option('display.width', 60)
         display(mat_view.df.filter(col('symbol') == 'TARO').limit(5).toPandas())
```

| | symbol | event_ts | trade_dt | price | bid | offer | ind_cd | epoch_ts | rolling_mean_price_lag_60 | rolling_mean_price_lag_300 | rolling_mean_price_lag_600 | rolling_mean_price_lag_1200 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | TARO | 2017-08-30 20:00:46 | 2017-08-31 | 346.499931 | None | None | 1 | 1504137646 | 346.499931 | 346.499931 | 346.499931 | 346.499931 |
| 1 | TARO | 2017-08-30 20:19:48 | 2017-08-31 | 348.398047 | None | None | 1 | 1504138788 | 348.398047 | 348.398047 | 348.398047 | 347.448989 |
| 2 | TARO | 2017-08-30 20:37:06 | 2017-08-31 | 346.411332 | None | None | 1 | 1504139826 | 346.411332 | 346.411332 | 346.411332 | 347.404689 |
| 3 | TARO | 2017-08-30 20:42:52 | 2017-08-31 | 349.811785 | None | None | 1 | 1504140172 | 349.811785 | 349.811785 | 348.111559 | 348.111559 |
| 4 | TARO | 2017-08-30 | 2017-08-31 | 347.540960 | None | None | 1 | 1504140432 | 347.540960 | 348.676373 | 348.676373 | 347.921359 |

마지막으로 로컬 IDE를 사용하는 동시에, 가장 규모가 큰 시계열 데이터 세트에 대해 구현한 시계열 뷰와 연결하면 두 가지의 장점을 모두 활용할 수 있습니다.

## 시장 조작에 Koalas 활용

pandas API는 Python에서 데이터 조작과 분석에 사용하는 표준 도구이며, Python 데이터 사이언스 에코시스템(예: NumPy, SciPy, Matplotlib)에 깊게 통합됩니다. pandas에 한 가지 단점이 있다면, 대량의 데이터로 쉽게 확장할 수 없다는 것입니다. 금융 데이터에는 언제나 위험 집계 또는 규정 준수 분석에 중요한 수년간의 과거 데이터가 포함되어 있습니다. 이 작업을 쉽게 처리하기 위해 Koalas를 백엔드에서 Spark를 실행하는 동안 pandas API를 활용할 도구로 삼았습니다. Koalas API는 pandas와 일치하기 때문에 여전히 사용하기 편리할 뿐만 아니라, 확장할 수 있는 코드로 옮기려면 한 줄의 코드만 변경하면 됩니다(다음 섹션의 Koalas 가져오기 참조). Koalas가 금융 시계열 문제에 적합한지 알아보기에 앞서, 금융 사기인 '주식 선매매 거래' 문제에 대해 간단히 알아보겠습니다.

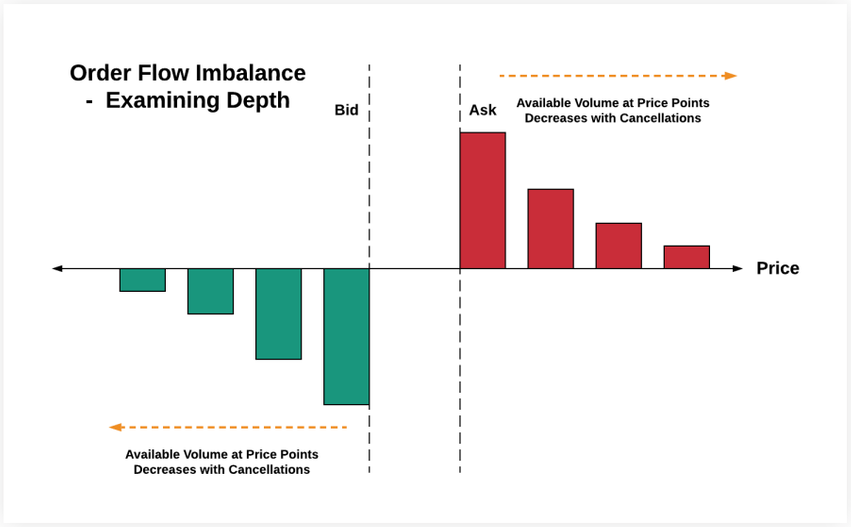
주식 선매매 거래는 다음과 같은 사건이 순차적으로 발생했을 때 일어납니다.

1. 거래 기업이 주가에 영향을 미칠 만한 비공개 정보를 알고 있습니다.
2. 이 기업이 대량의 주문을 냅니다(또는 모두 합치면 대량이 되는 주문을 몇 번에 걸쳐 대규모로 냅니다).
3. 유동성이 제거되어 주가가 상승합니다.
4. 이 기업이 주식을 (이전보다 훨씬 상승한 가격으로) 투자자에게 매도하여 큰 이익을 챙기고, 투자자는 주식 거래를 발생시킨 정보가 비공개였음에도 불구하고 더 높은 가격을 지불해야 합니다.

출처: CC0 공개 도메인 이미지 - **왼쪽** 이미지, **오른쪽** 이미지

참고하실 수 있도록 농산물 시장과 애플파이 기업에 대한 간단한 예시를 **이곳**에서 제공합니다. 이 예시에는 Freddy가 등장합니다. Freddy는 전국의 애플파이 사업에 사과가 즉시 필요하다는 것을 알고 모든 농산물 시장에서 사과를 구매합니다. 이런 행동으로 인해 Freddy는 다른 구매자(투자자)가 제품을 구매하기 전에 누구보다 먼저 구매하여 큰 영향을 미치고, 구매자에게 프리미엄을 붙여 사과를 판매할 수 있게 됩니다.

주식 선매매 거래를 탐지하려면 주문 흐름의 불균형을 알아내야 합니다 (아래의 그림 참조). 특히, 주문 흐름 불균형의 이상치는 주식 선매매 거래가 발생할 만한 기간을 알아내는 데 도움이 됩니다.




이제 Koalas 패키지를 사용하여 생산성을 개선하고 시장 조작 문제를 해결해보겠습니다. 즉, 다음에 초점을 맞춰 주문 흐름 불균형 이상치를 찾아보겠습니다.

- 동시 이벤트의 중복 제거
- 공급/수요 증가 평가를 위한 시차 윈도
- 데이터 프레임을 병합하여 주문 흐름 불균형 집계

**시계열의 중복 제거**

일반적 시계열 데이터 정제에는 대치(imputation)와 중복 제거(de-duplication)를 사용합니다. 빈도가 높은 데이터(예: 호가 데이터)에서는 값이 중복될 수 있습니다. 순차 번호가 없고 시간당 여러 개의 값이 있는 경우, 중복을 제거해서 이후의 통계 분석에 문제가 없도록 해야 합니다. 아래의 사례에서는 시점당 여러 매수/매도 주식이 보고되는데, 주문 불균형을 계산하려면 시간당 최대 깊이에 대해 하나의 값만 사용해야 합니다.

```
import databricks.koalas as ks

kdf_src = ks.read_delta("...")
grouped_kdf = kdf_src.groupby(['event_ts'],
as_index=False).max()
grouped_kdf.sort_values(by=['event_ts'])
grouped_kdf.head()
```

| | Symbol | Date | Time | bid_pr | ask_pr | bid_shrs_qt | ask_shrs_qt | event_ts |
|---|---|---|---|---|---|---|---|---|
| **39757** | ITUB | 03/05/2014 | 09:30:00.011 | 13.14 | 13.23 | 700.0 | 200.0 | 2014-03-05 09:30:00.011 |
| **39758** | ITUB | 03/05/2014 | 09:30:00.052 | 13.15 | 13.23 | 700.0 | 200.0 | 2014-03-05 09:30:00.052 |
| **39759** | ITUB | 03/05/2014 | 09:30:00.235 | 13.15 | 13.22 | 700.0 | 100.0 | 2014-03-05 09:30:00.235 |
| **39760** | ITUB | 03/05/2014 | 09:30:00.236 | 13.16 | 13.22 | 100.0 | 100.0 | 2014-03-05 09:30:00.236 |
| **39761** | ITUB | 03/05/2014 | 09:30:00.237 | 13.16 | 13.21 | 100.0 | 700.0 | 2014-03-05 09:30:00.237 |

### Koalas를 사용한 시계열 윈도잉

시계열에서 중복을 제거했으므로 윈도우에서 수요와 공급을 찾아보겠습니다. 일반적으로 시계열에서 윈도잉(windowing)이란 슬라이스나 시간 간격을 살펴보는 것입니다. 대부분 추세 계산(예: 단순 이동평균)에서는 모두 타임 윈도우의 개념을 사용하여 연산합니다. Koalas는 아래와 같이 `shift` (Spark의 lag 함수와 유사)를 사용하여 하나의 윈도우 내의 선행(lead) 또는 후행(lag) 값을 가져오는 단순 pandas 인터페이스를 상속합니다.

```
grouped_kdf.set_index('event_ts', inplace=True, drop=True)
lag_grouped_kdf = grouped_kdf.shift(periods=1, fill_value=0)

lag_grouped_kdf.head()
```

| event_ts | Symbol | Date | Time | bid_pr | ask_pr | bid_shrs_qt | ask_shrs_qt |
|---|---|---|---|---|---|---|---|
| **2014-03-05 09:30:00.011** | 0 | 0 | 0 | 0 | 0 | 0.0 | 0.0 |
| **2014-03-05 09:30:00.052** | ITUB | 03/05/2014 | 09:30:00.011 | 13.14 | 13.23 | 700.0 | 200.0 |
| **2014-03-05 09:30:00.235** | ITUB | 03/05/2014 | 09:30:00.052 | 13.15 | 13.23 | 700.0 | 200.0 |
| **2014-03-05 09:30:00.236** | ITUB | 03/05/2014 | 09:30:00.235 | 13.15 | 13.22 | 700.0 | 100.0 |
| **2014-03-05 09:30:00.237** | ITUB | 03/05/2014 | 09:30:00.236 | 13.16 | 13.22 | 100.0 | 100.0 |

### 타임스탬프에 병합 및 Koalas 열 연산으로 불균형 계산

후행(lag) 값을 계산했으므로 원래 시계열 호가와 이 데이터 세트를 병합해야 합니다, 아래에서는 Koalas `merge` 를 사용하여 시간 색인을 적용합니다. 그러면 공급/수요 계산에 필요한 통합 뷰가 생성되고, 주문 불균형 지표를 알 수 있습니다.

```
lagged = grouped_kdf.merge(lag_grouped_kdf, left_index=True, right_index=True,
suffixes=['', '_lag'])
lagged['imblnc_contrib'] = lagged['bid_shrs_qt']*lagged['incr_demand'] \
    - lagged['bid_shrs_qt_lag']*lagged['decr_demand'] \
    - lagged['ask_shrs_qt']*lagged['incr_supply'] \
    + lagged['ask_shrs_qt_lag']*lagged['decr_supply']
```

| event_ts | Symbol | Time | bid_pr | ask_pr | bid_pr_lag | ask_pr_lag | imblnc_contrib |
|---|---|---|---|---|---|---|---|
| **2014-03-05 09:30:00.011** | ITUB | 09:30:00.011 | 13.14 | 13.23 | 0 | 0 | 500.0 |
| **2014-03-05 09:30:00.052** | ITUB | 09:30:00.052 | 13.15 | 13.23 | 13.14 | 13.23 | 0.0 |
| **2014-03-05 09:30:00.235** | ITUB | 09:30:00.235 | 13.15 | 13.22 | 13.15 | 13.23 | 100.0 |
| **2014-03-05 09:30:00.236** | ITUB | 09:30:00.236 | 13.16 | 13.22 | 13.15 | 13.22 | -600.0 |
| **2014-03-05 09:30:00.237** | ITUB | 09:30:00.237 | 13.16 | 13.21 | 13.16 | 13.22 | -600.0 |

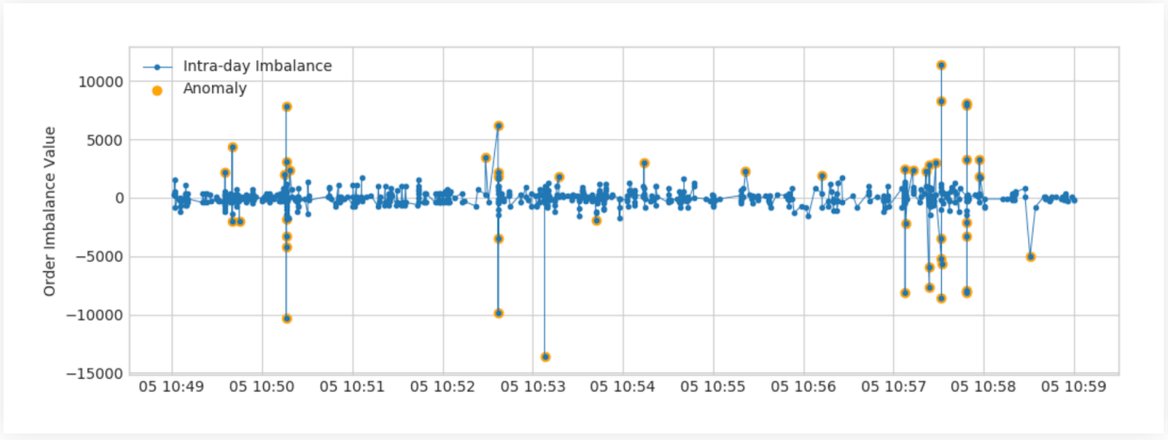
### Koalas-NumPy 변환으로 분포 피팅

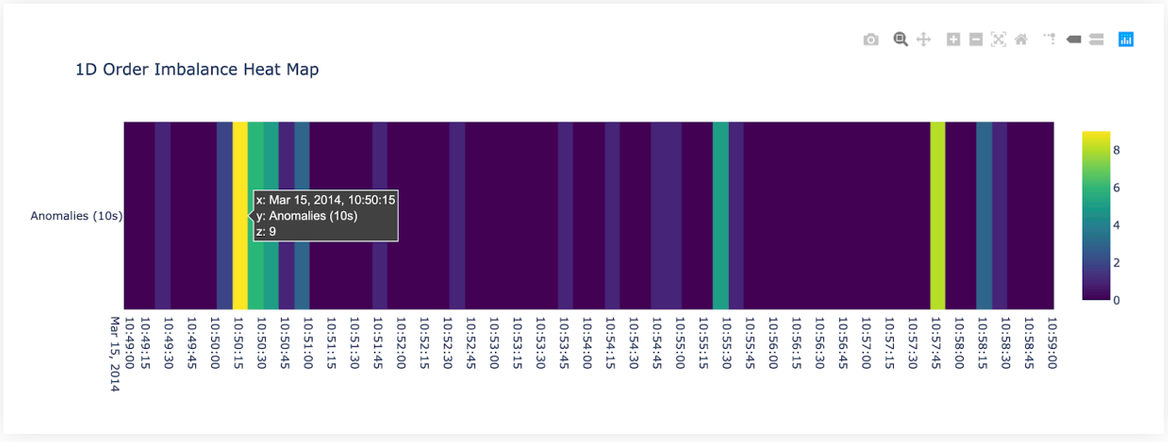
초기 준비가 끝났으므로 Koalas 데이터 프레임을 통계 분석에 유용한 형식으로 바꾸어야 합니다. 이 문제의 경우, 분석을 진행하기 전에 분 또는 다른 시간 단위로 불균형을 집계할 수 있지만 참고로 보여드리기 위해 티커 “ITUB”에 대해 전체 데이터 세트를 실행할 것입니다. Koalas 구조를 NumPy 데이터 세트로 변환하고, SciPy 라이브러리를 사용하여 주문 흐름 불균형에서 이상치를 탐지하는 방법은 다음과 같습니다. to_numpy() 구문을 사용하여 이 분석을 연결합니다.

```
from scipy.stats import t
import scipy.stats as st
import numpy as np

q_ofi_values = lagged['imblnc_contrib'].to_numpy()
```

아래에서 주문 흐름 불균형 분포를 나타낸 그래프에 5번째 백분위수와 95번째 백분위수의 마커를 표시하여 불균형 이상이 발생한 시점의 이벤트를 확인할 수 있도록 했습니다. 분포를 피팅하고 이 그래프를 그리기 위한 코드는 전체 노트북을 참조하세요. Koalas/SciPy 워크플로로 계산한 불균형이 발생한 시점은 우리가 찾고자 했던 시장 조작 사기인 주식 선매매가 발생할 가능성이 있는 기간을 나타냅니다.

아래의 시계열 시각화는 위의 이상치로 검색된 이상을 점으로 나타냈으며, 주황색으로 강조했습니다. 최종 시각화에서는 `plotly` 라이브러리를 사용하여 열지도의 형태로 이상이 발생하는 기간과 빈도를 요약했습니다. 특히, **10:50:10-10:50:20** 시간은 주식 선매매가 발생했을 가능성이 있는 구간입니다.




## 결론

이 문서에서는 Apache Spark과 Databricks에서 윈도잉과 래퍼를 사용하여 직접적으로 시계열 분석에 활용하고, Koalas를 사용하여 간접적으로 활용하는 방법을 알아보았습니다. 대부분 데이터 사이언티스트는 pandas API를 사용하므로 Koalas는 Apache Spark의 규모에서 pandas 기능을 사용하는 데 도움을 줍니다. 시계열 분석에 Spark와 Koalas를 사용하는 데는 다음과 같은 장점이 있습니다.

- 조인 기준 및 단순 집계를 사용하여 위험, 사기, 규정 준수 사용 사례에 대한 시계열 분석 병렬화
- Databricks Connect를 사용한 빠른 반복 개선과 리치 시계열 feature 생성
- 데이터 사이언스 및 퀀트팀에게 Koalas를 제공하여 pandas와 API를 편리하게 활용하면서도 데이터 준비 규모를 확대할 수 있도록 지원

Databricks에서 이 **노트북**을 확인해보세요! 금융 시계열 사용 사례와 관련하여 고객에게 어떤 도움을 드리는지 알고 싶다면 당사에 문의해주세요.

> 이 무료 Databricks **노트북**으로 실험을 시작하세요.

3장:

# 동적 시간 왜곡의 이해

**1부 동적 시간 왜곡 및 MLflow를 사용한 매출 동향 발견 시리즈**

글: Ricardo Portilla, Brenner Heintz 및 Denny Lee

2019년 4월 30일

Databricks에서 이 노트북을 확인해보세요 →

## 소개

"동적 시간 왜곡(dynamic time warping)"이라는 표현을 처음 읽으면 <백투더퓨처> 시리즈에서 주인공 마티 맥플라이가 드로리언을 시속 142km로 달리던 모습을 떠올릴 수도 있습니다. 아, 동적 시간 왜곡은 시간 여행과는 관계가 없습니다. 비교 데이터 포인트 사이의 시간 색인이 완전히 동기화되지 않을 때 동적으로 시계열을 비교하는 데 사용하는 기술입니다.

나중에 설명해 드리겠지만 동적 시간 왜곡의 가장 핵심적인 용도는 음성 인식입니다. 어떤 구절을 빠르거나 느리게 말하더라도 다른 구절과 일치하는지 알아내는 데 사용합니다. Google Home이나 Amazon Alexa 기기를 활성화시키는 "깨우는 말(wake words)"을 인식할 때 얼마나 유용할지 상상이 가시죠? 모닝커피를 마시지 못해서 다소 말이 어눌하게 나오더라도 인식할 수 있습니다.

동적 시간 왜곡은 다양한 분야에 적용할 수 있는 유용하고 효과적인 기술입니다. 동적 시간 왜곡의 개념을 이해했다면 일상생활에서 활용되는 사례를 쉽게 발견할 수 있고, 앞으로도 기대가 큰 응용 분야입니다. 다음과 같은 사례를 생각해볼 수 있습니다.

- **금융 시장:** 비슷한 기간에 대해 완벽하게 일치하지 않는 증권 거래 데이터를 비교합니다. 예를 들어 2월(28일)과 3월(31일)에 대한 월간 거래 데이터를 비교하는 것입니다.
- **웨어러블 피트니스 트래커:** 사용자가 걷는 속도가 시간에 따라 달라지더라도 걷는 속도와 걸음 수를 정확히 계산합니다.
- **경로 계산:** 운전 습관에 대해 알고 있을 경우(예: 직진 도로에서는 빠르게 운전하지만 좌회전할 때는 평균보다 시간이 더 걸리는 습관) 운전자의 ETA에 대해 보다 정확한 정보를 계산합니다.

데이터 사이언티스트, 데이터 분석가, 시계열을 사용해서 일하는 사람이라면 누구나 이 기술을 알고 있어야 합니다. 완벽하게 일치하는 시계열 비교 데이터는 실전에서 완벽하게 "깔끔한" 데이터만큼이나 보기 어렵기 때문입니다.

블로그 시리즈에서 다룰 내용:

- 동적 시간 왜곡의 기본 원칙
- 샘플 오디오 데이터에서 동적 시간 왜곡 실행
- MLflow를 사용하여 샘플 매출 데이터에서 동적 시간 왜곡 실행

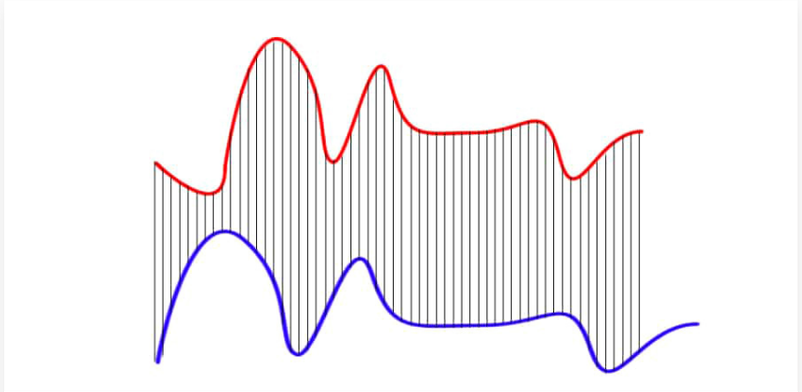
## 동적 시간 왜곡

시계열 비교법의 목표는 두 입력 시계열 사이에 *거리 척도*를 만드는 것입니다. 일반적으로 두 시계열의 유사성이나 비유사성은 데이터를 벡터로 변환하여 벡터 공간에서 두 지점 사이의 유클리드 거리(직선거리)를 계산하여 산출합니다.

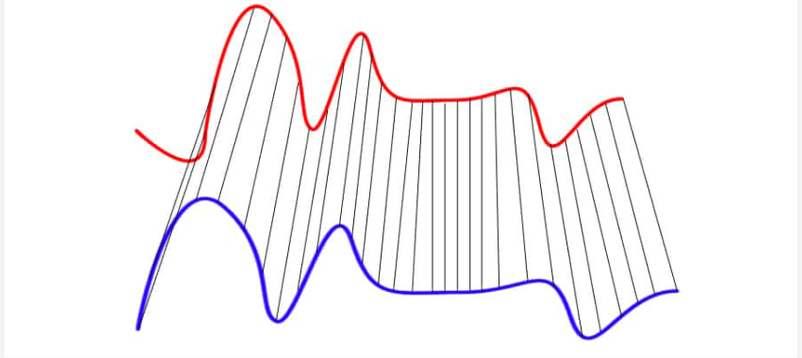
동적 시간 왜곡은 1970년대에 음파 소스에서 음성과 단어를 인식하는 데 사용하던 획기적인 시계열 비교 기법입니다. 주로 인용되는 논문은 "Dynamic time warping for isolated word recognition based on ordered graph searching techniques 입니다."

### 배경

이 기술은 패턴 매칭뿐만 아니라, 이상 탐지에도 사용할 수 있습니다(예: 두 개의 연속적이지 않은 기간 사이의 시계열을 중첩해서 그 형태가 크게 변화하는지 확인하거나 이상치를 찾습니다). 예를 들어, 이 그래프에서 빨간색과 파란색 선을 보면 기존 시계열 매칭 (유클리드 매칭)은 매우 제한적입니다. 반면, 동적 시간 왜곡을 사용하면 X축(즉, 시간)이 동기화되지 않았더라도 두 곡선이 두 곡선이 균등하게 매칭됩니다. 이 방법은 로버스트 상이도 점수로 생각해도 됩니다. 숫자가 낮을수록 시계열이 더욱 유사한 것을 의미합니다.



유클리드 매칭



동적 시간 왜곡 매칭

출처: Wikimedia Commons
File: Euclidean_vs_DTW.jpg

두 시계열(기준 시계열 및 새로운 시계열)은 다음의 규칙을 따르는 함수 f(x)로 매핑해서 최적(왜곡) 경로를 사용하여 크기를 매칭할 수 있을 때 유사한 것으로 간주합니다.

$$f(x_i) \text{ maps to } f(x_j) \text{ when } i <= j$$

$$f(x_i) \text{ maps to } f(x_j) \text{ only when } (j - i) \text{ is within fixed range}$$

**사운드 패턴 매칭**

원래 동적 시간 왜곡은 오디오 클립에 적용하여 해당 클립의 유사성을 알아내는 데 사용했습니다. 우리 예시에서는 "The Expanse" 라는 TV 드라마의 대사 두 가지를 녹음한 오디오 클립 4개를 사용하겠습니다. 오디오 클립은 4개가 있습니다 (아래에서 들어볼 수는 있지만 들어보지 않으셔도 됩니다). 그중 3개(클립 1, 2, 4)의 대사는 다음과 같습니다.

***"문과 구석진 곳, 그런 데서 당하는 거야"라는 대사가 나오는 기본 시계열입니다.***

나머지 1개(클립 3)의 대사는 다음과 같습니다.

***"방 안으로 너무 빨리 들어가면, 방에 잡아먹히지."***

Clip 1 문과 구석진 곳, 그런 데서 당하는 거야. [v1]

0:00 / 0:06

Clip 2 문과 구석진 곳, 그런 데서 당하는 거야. [v2]

0:00 / 0:08

Clip 3 방 안으로 너무 빨리 들어가면, 방에 잡아먹히지.

0:00 / 0:07

Clip 4 문과 구석진 곳, 그런 데서 당하는 거야. [v3]

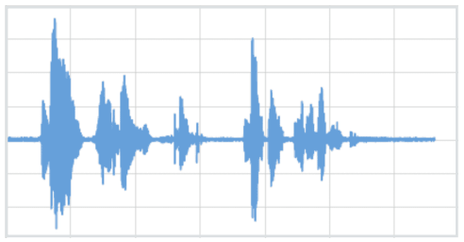
0:00 / 0:07

"The Expanse"의 대사

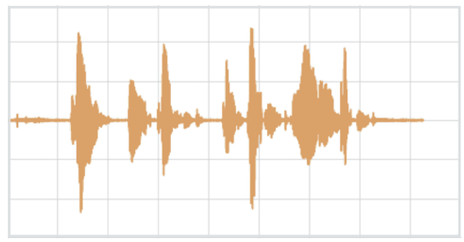
다음은 4개의 오디오 클립에 `matplotlib` 을 사용하여 시각화한 결과입니다.

- **클립 1:** "문과 구석진 곳. 그런 데서 당하는 거야"라는 대사가 나오는 기본 시계열입니다.
- **클립 2:** 클립 1에 기반한 새로운 시계열[v2]로, 어조와 말하는 패턴이 매우 과장되어 있습니다.
- **클립 3:** 클립 1과 동일한 어조와 음성이지만 "방 안으로 너무 빨리 들어가면, 방에 잡아먹히지"라는 대사가 나오는 시계열입니다.
- **클립 4:** 클립 1에 기반한 새로운 시계열[v3]로, 어조와 말하는 패턴이 클립 1과 유사합니다.

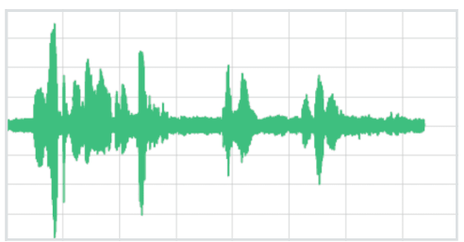
Clip 1 문과 구석진 곳, 그런 데서 당하는 거야. [v1]



Clip 2 문과 구석진 곳, 그런 데서 당하는 거야. [v2]



Clip 3 방 안으로 너무 빨리 들어가면, 방에 잡아먹히지.



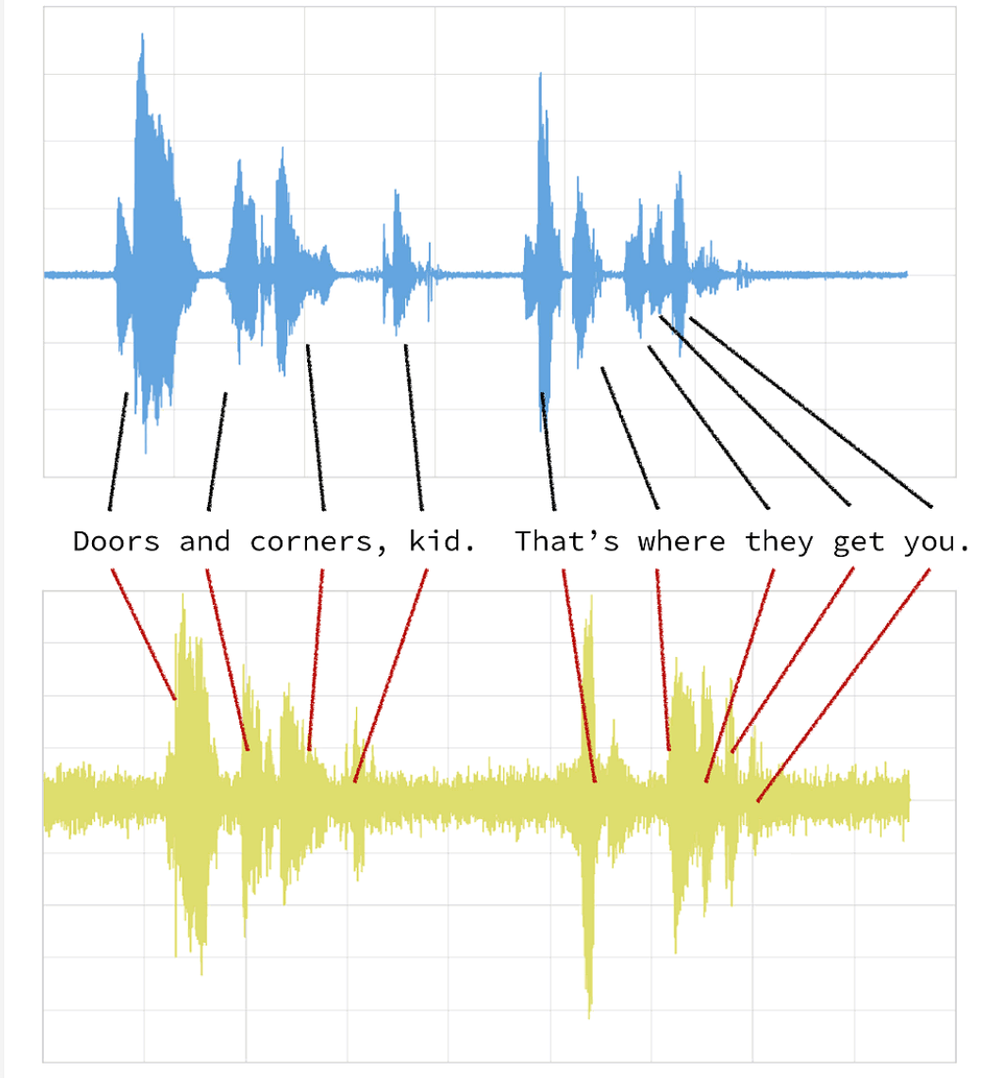
Clip 4 문과 구석진 곳, 그런 데서 당하는 거야. [v3]

오디오 클립을 읽고 matplotlib을 사용하여 시각화하는 코드는 다음의 코드 조각으로 요약할 수 있습니다.

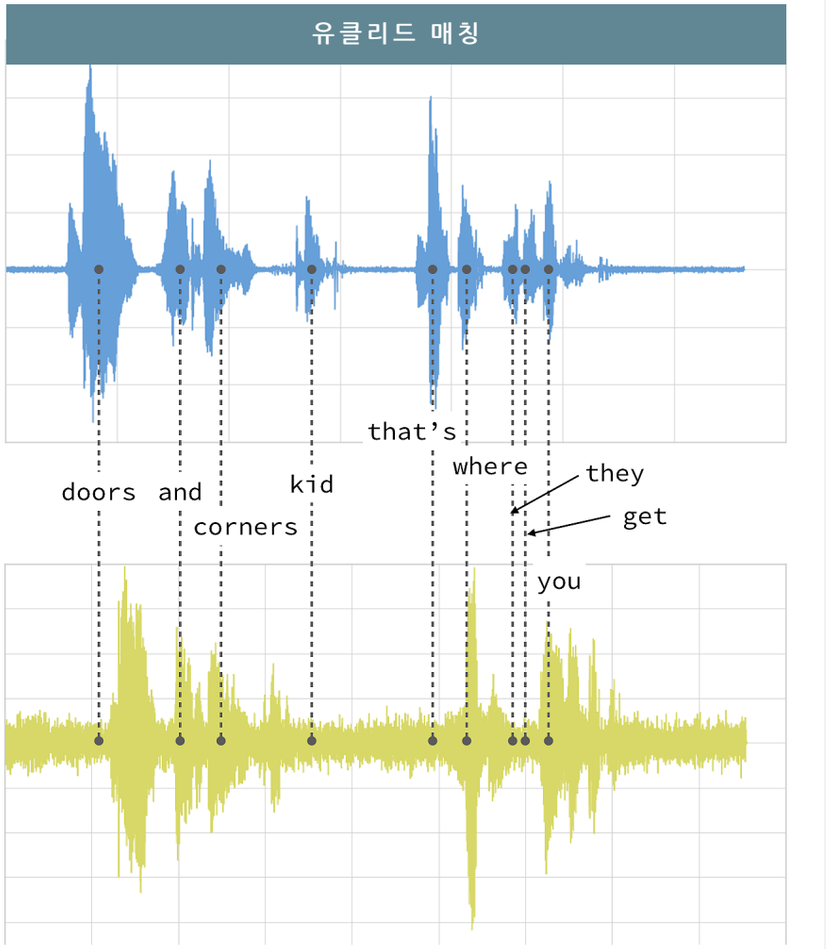
```
from scipy.io import wavfile
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure

# Read stored audio files for comparison
fs, data = wavfile.read("/dbfs/folder/clip1.wav")

# Set plot style
plt.style.use('seaborn-whitegrid')

# Create subplots
ax = plt.subplot(2, 2, 1)
ax.plot(data1, color='#67A0DA')
...

# Display created figure
fig=plt.show()
display(fig)
```

전체 코드베이스는 **동적 시간 왜곡 배경** 노트북에 나와 있습니다.

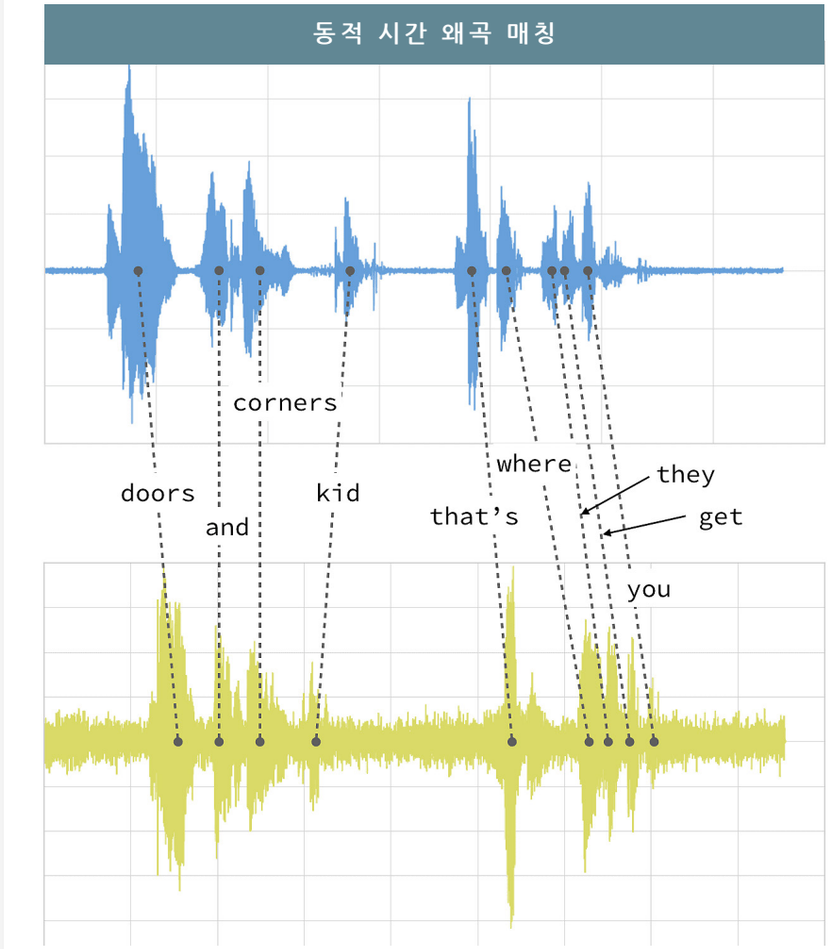
아래에서 설명했듯이, 두 클립(이 경우 클립 1과 4)은 같은 대사지만 어조(진폭)와 지연 시간이 다릅니다.



기존 유클리드 매칭(아래 그래프 참조)을 따를 경우, 진폭을 무시하더라도 원본 클립(파란색)과 새로운 클립(노란색) 간의 타이밍이 일치하지 않습니다.




동적 시간 왜곡을 적용하면 시간을 옮겨서 두 클립의 시계열을 비교할 수 있습니다.

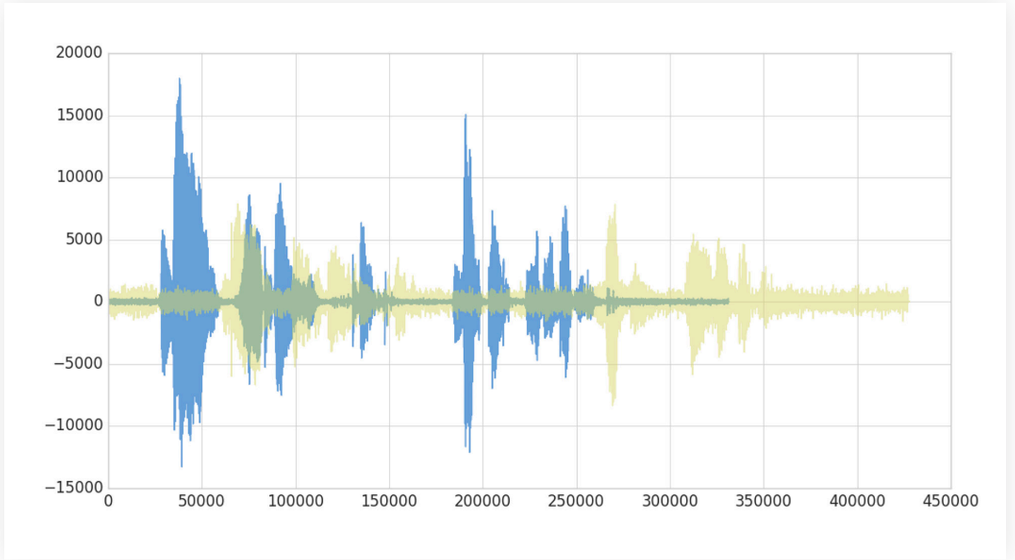


이 시계열 비교에서는 `fastdtw` PyPi 라이브러리를 사용합니다. Databricks 작업 영역에 PyPi 라이브러리를 설치하는 명령은 Azure | AWS에서 확인할 수 있습니다. fastdtw를 사용하면 두 시계열 사이의 거리를 빠르게 계산할 수 있습니다.

```
from fastdtw import fastdtw

# Distance between clip 1 and clip 2
distance = fastdtw(data_clip1, data_clip2)[0]
print("The distance between the two clips is %s" % distance)
```

전체 코드 베이스는 **동적 시간 왜곡 배경** 노트북에 나와 있습니다.

| 기준 | 쿼리 | 거리 |
|---|---|---|
| 클립 1 | 클립 2 | 480148446.0 |
| | 클립 3 | 310038909.0 |
| | 클립 4 | 293547478.0 |




관찰 결과 요약:

- 앞서 보여드린 그래프에서 확인하였듯이, 클립 1과 4는 어조와 단어가 동일하므로 가장 거리가 짧습니다.
- 클립 1과 3 사이의 거리도 (클립 4에 비해 길기는 하지만) 상당히 짧습니다.
  단어는 다르지만 어조가 동일하고 말하는 속도가 같습니다.
- 클립 1과 2는 같은 대사이기는 하지만 어조와 말하는 속도가 매우 과장되어 있어서 거리가 가장 깁니다.

여러분도 보다시피, 동적 시간 왜곡을 사용하면 두 시계열의 유사성을 알아낼 수 있습니다.

## 다음

동적 시간 왜곡에 관해 설명해드렸으므로 이 사용 사례를 **매출 동향을 탐지**하는 데 적용해보겠습니다.

3장:

# 동적 시간 왜곡 및 MLflow를 사용한 매출 동향 발견

2부 동적 시간 왜곡 및 MLflow를 사용한 매출 동향 발견 시리즈

글: Ricardo Portilla, Brenner Heintz 및 Denny Lee

2019년 4월 30일

Databricks에서 이 노트북 시리즈(DBC 형식)를 확인해보세요 →

## 배경

여러분이 3D 프린팅 제품을 제작하는 회사를 소유한 사장이라고 생각해보세요. 작년에 드론 프로펠러의 수요가 매우 꾸준하다는 것을 알았기 때문에 그 제품을 판매했습니다. **그 전 해에는 휴대전화 케이스를 판매했습니다. 조만간 새해를 맞이하기에, 제조팀과 내년에 생산할 제품에 대해 협의하고 있습니다.** 3D 프린터를 구매하려면 많은 대출을 받아야 하기 때문에 대출을 상환하려면 구매한 프린터를 항상 100%나 그에 가깝게 사용해야 합니다.

여러분은 현명한 CEO이기 때문에 내년 생산 능력이 변동할 수밖에 없다는 것을 알고 있고 생산 능력이 유난히 높은 주도 있다는 것을 압니다. 예를 들어 여름에 (임시 근로자를 채용하면), 생산 능력이 높아지고 매월 셋째 주에는 (3D 프린터 필라멘트 공급망 문제로 인해) 생산 능력이 저하됩니다. 아래의 그래프는 회사의 생산 능력 추산치입니다.


databricks

**주간 수요가 생산 능력과 최대한 가까운 제품을 선택하는 것이 목표입니다.** 각 제품에 대한 작년 매출을 포함한 카탈로그를 살펴보고 있는데, 올해 매출도 비슷할 것으로 예상합니다.

주간 수요가 자신의 생산 능력을 초과하는 제품을 선택할 경우, 고객 주문을 취소해야 하므로 사업에 좋지 못합니다. 반면, 주간 수요가 충분하지 않은 제품을 선택한다면 프린터의 사용률을 최대로 높일 수 없어서 대출을 상환하지 못할 수도 있습니다.

**이럴 때 동적 시간 왜곡을 사용합니다. 선택한 제품의 공급과 수요가 다소 일치하지 않을 수 있기 때문입니다.** 모든 수요를 감당할 수 있을 만큼 생산 여력이 충분하지 않은 주도 있겠지만, 차이가 지나치게 크지 않다면 그 주보다 한두 주 전후로 제품을 더 많이 생산해서 보충한다면 고객에게는 영향이 미치지 않을 것입니다. 유클리드 매칭을 사용해서 매출 데이터와 생산 능력을 비교하는 데 그친다면, 이를 고려하지 않은 제품을 선택해서 돈을 벌 기회를 놓치게 됩니다. 대신 우리는 동적 시간 왜곡을 사용해서 올해 회사 상황에 맞는 제품을 선택할 것입니다.

## 제품 매출 데이터 세트 로드

**UCI 데이터 세트 리포지토리**에 있는 **주간 매출 거래 데이터 세트**를 사용해서 매출 기반 시계열 분석을 적용합니다. (출처: James Tan, **jamestansc@suss.edu.sg**, Singapore University of Social Sciences)

```
import pandas as pd

# Use Pandas to read this data
sales_pdf = pd.read_csv(sales_dbfspath, header='infer')

# Review data
display(spark.createDataFrame(sales_pdf))
```

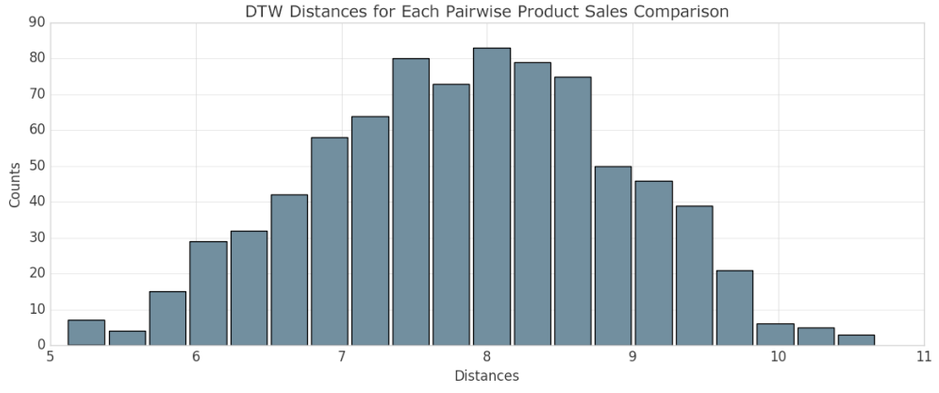
| Product_Code | W0 | W1 | W2 | W3 | W4 | W5 | W6 | W7 | W8 | W9 | W10 | W11 | W12 | W13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P1 | 11 | 12 | 10 | 8 | 13 | 12 | 14 | 21 | 6 | 14 | 11 | 14 | 16 | 9 |
| P2 | 7 | 6 | 3 | 2 | 7 | 1 | 6 | 3 | 3 | 3 | 2 | 2 | 6 | 2 |
| P3 | 7 | 11 | 8 | 9 | 10 | 8 | 7 | 13 | 12 | 6 | 14 | 9 | 4 | 7 |
| P4 | 12 | 8 | 13 | 5 | 9 | 6 | 9 | 13 | 13 | 11 | 8 | 4 | 5 | 4 |
| P5 | 8 | 5 | 13 | 11 | 6 | 7 | 9 | 14 | 9 | 9 | 11 | 18 | 8 | 4 |
| P6 | 3 | 3 | 2 | 7 | 6 | 3 | 8 | 6 | 6 | 3 | 1 | 1 | 5 | 4 |
| P7 | 4 | 8 | 3 | 7 | 8 | 7 | 2 | 3 | 10 | 3 | 5 | 2 | 3 | 4 |
| P8 | 8 | 6 | 10 | 9 | 6 | 8 | 7 | 5 | 10 | 10 | 8 | 8 | 15 | 9 |

각 제품은 행으로 나타내고 올해의 각 주는 열로 나타냅니다. 값은 매주 판매되는 각 제품의 단위를 나타냅니다. 데이터 세트에는 811개 제품이 있습니다.

## 제품 코드별 최적 시계열과의 거리 계산

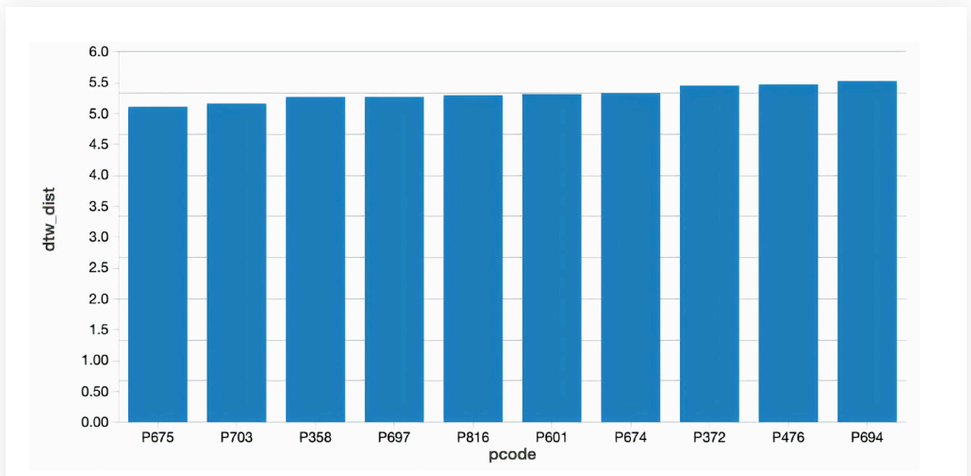
```
# Calculate distance via dynamic time warping between product code and
optimal time series
import numpy as np
import _ucrdtw

def get_keyed_values(s):
    return(s[0], s[1:])

def compute_distance(row):
    return(row[0], _ucrdtw.ucrdtw(list(row[1][0:52]), list(optimal_pattern),
0.05, True)[1])

ts_values = pd.DataFrame(np.apply_along_axis(get_keyed_values, 1, sales_pdf.
values))
distances = pd.DataFrame(np.apply_along_axis(compute_distance, 1, ts_values.
values))
distances.columns = ['pcode', 'dtw_dist']
```

계산된 동적 시간 왜곡 '거리' 열을 사용하면 DTW 거리의 분포를 히스토그램으로 확인할 수 있습니다.

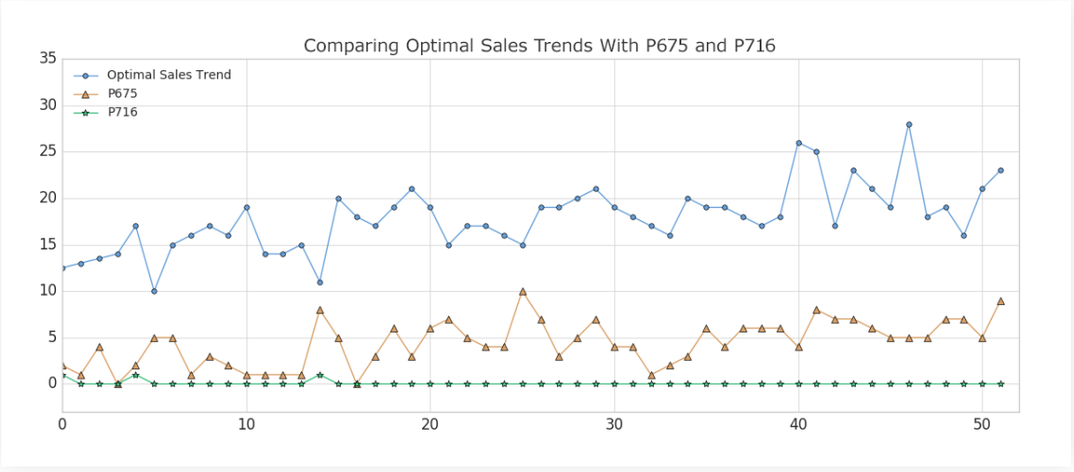


여기에서 최적 매출 동향과 가장 가까운 제품 코드를 찾을 수 있습니다(즉, 산출된 DTW 거리가 가장 작은 제품). 우리는 Databricks를 사용하고 있기 때문에 SQL 쿼리로 간편하게 선택할 수 있습니다. DTW 거리가 가장 가까운 제품을 표시해보겠습니다.

```
%sql
-- Top 10 product codes closest to the optimal sales trend
select pcode, cast(dtw_dist as float) as dtw_dist from distances order by cast(dtw_dist as float) limit 10
```




이 쿼리를 실행한 뒤, 최적 매출 동향에서 가장 멀리 *떨어진* 제품 코드에 대한 해당 쿼리를 실행해서 매출 동향으로부터 거리가 가장 가까운 제품과 가장 먼 제품 두 개를 찾았습니다. 그래프를 통해 두 제품에 어떤 차이가 있는지 살펴보겠습니다.

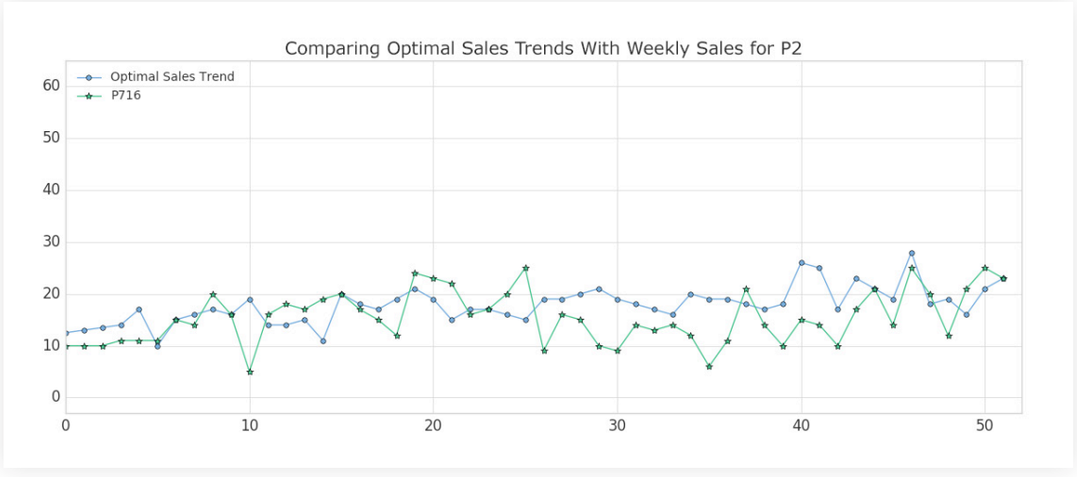



여러분도 볼 수 있듯이, 제품 #675(주황색 삼각형)는 최적 매출 동향과 가장 일치하지만 절대 주간 매출은 우리가 원하는 수준보다 낮습니다(이는 나중에 해결하겠습니다). 이런 결과를 이해할 수 있는 이유는 DTW 거리가 가장 가까운 제품은 비교 대상 지표를 다소 반영하여 고점과 저점이 있을 것으로 예상했기 때문입니다. (물론, 제품의 정확한 시간 색인은 동적 시간 왜곡으로 인해 매주 다릅니다). 반면, 제품 #716(녹색 별)은 가장 일치하지 않는 제품이라 거의 분산이 없습니다.

## 최적 제품 찾기: DTW 거리가 가깝고, 절대 매출이 유사한 제품

공장의 예상 생산량(“최적 매출 동향”)과 가장 가까운 제품 목록을 만들었으므로 DTW 거리가 가까우면서도 절대 매출이 유사한 제품만 필터링해보겠습니다. 가능성이 있는 후보로는 제품 #202가 있습니다. DTW 거리가 모집단 중간값은 7.89인데 비해 6.86이고, 최적 매출 동향을 매우 근사하게 따라갑니다.

```
# Review P202 weekly sales
y_p202 = sales_pdf[sales_pdf['Product_Code'] == 'P202'].values[0][1:53]
```




## MLflow를 사용하여 아티팩트와 함께 가장 적절한 제품과 가장 부적절한 제품 추적

MLflow는 실험, 재현, 배포를 포함한 머신 러닝 수명 주기를 관리하기 위한 오픈 소스 플랫폼입니다. **Databricks 노트북은 완전 통합 MLflow 환경을 제공하므로, 실험을 생성하여 매개변수와 지표를 기록하고 결과를 저장할 수 있습니다.** 이 유용한 문서에서 MLFlow를 시작하는 방법에 대한 자세한 정보를 확인할 수 있습니다.

**MLflow는 각 실험의 모든 입력값과 출력값을 체계적이고 재현할 수 있는 방식으로 기록하는 기능을 중심으로 설계됩니다.** “런”이라고 하는 데이터를 통과시키는 과정을 실행할 때마다 실험값을 기록할 수 있습니다.

- **매개변수**: 모델에 대한 입력값
- **지표**: 모델의 출력값, 모델의 성공 측정
- **아티팩트**: 모델에서 생성한 모델 — 예: PNG 그래프, 또는 CSV 데이터 결과값
- **모델**: 모델 자체(나중에 다시 로드에서 예측하는 데 사용 가능)

우리 사례에서는 “신장 계수(stretch factor)”를 변경하면서 데이터에 동적 시간 왜곡 알고리즘을 여러 번 실행할 수 있습니다. 신장 계수란 시계열 데이터에 적용할 수 있는 최대 왜곡 횟수를 나타냅니다. MLflow 실험을 초기화하고 `mlflow.log_param()`, `mlflow.log_metric()`, `mlflow.log_artifact()`, `mlflow.log_model()`를 사용하여 간편하게 로깅하기 위해 다음을 사용하여 메인 함수를 래핑했습니다.

```
iwith mlflow.start_run() as run:
    ...
```

요약된 코드는 아래와 같습니다.

```
import mlflow

def run_DTW(ts_stretch_factor):
    # calculate DTW distance and Z-score for each product
    with mlflow.start_run() as run:

        # Log Model using Custom Flavor
        dtw_model = {‘stretch_factor’ : float(ts_stretch_factor), ‘pattern’ : optimal_
pattern}
        mlflow_custom_flavor.log_model(dtw_model, artifact_path=”model”)

        # Log our stretch factor parameter to MLflow
        mlflow.log_param(“stretch_factor”, ts_stretch_factor)

        # Log the median DTW distance for this run
        mlflow.log_metric(“Median Distance”, distance_median)

        # Log artifacts - CSV file and PNG plot - to MLflow
        mlflow.log_artifact(‘zscore_outliers_’ + str(ts_stretch_factor) + ‘.csv’)
        mlflow.log_artifact(‘DTW_dist_histogram.png’)

    return run.info

stretch_factors_to_test = [0.0, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5]
for n in stretch_factors_to_test:
    run_DTW(n)
```

데이터에 ‘런’을 한 번 실행할 때마다 사용한 “신장 계수”의 로그와 DTW 거리 지표의 Z 점수를 기준으로 이상치로 분류한 Z-제품의 로그를 생성했습니다. 또한, DTW 거리 히스토그램의 아티팩트(파일)도 저장했습니다. **이 실험적 런은 Databricks에 로컬로 저장되고 나중에 실험 결과를 다시 보고 싶을 때 액세스할 수 있습니다.**

MLflow에서 각 실험의 로그를 저장했으므로 결과를 살펴볼 수 있습니다. Databricks 노트북에서 오른쪽 상단 모서리에 있는 “Runs” 아이콘을 선택하여 각 런의 결과를 확인하고 비교합니다.

당연하게도 “신장 계수”를 높일수록 거리 계수가 감소합니다. 직관적으로 생각해도 타당합니다. 시간 색인을 앞이나 뒤로 왜곡하도록 알고리즘에 유연성을 부여할수록 데이터에 더욱 가까운 결과를 찾아냅니다. 기본적으로는 분산 대신 편향을 감수한 것입니다.

## MLflow를 사용한 모델 로깅

MLflow는 실험 매개변수, 지표, 아티팩트(예: 그래프, CSV 파일)를 기록할 수 있을 뿐만 아니라, 머신 러닝 모델도 기록할 수 있습니다. MLflow 모델은 일관적 API를 구성하는 구조의 폴더로, 다른 MLflow 도구 및 기능과의 호환성을 보장합니다. 이런 상호운용성은 매우 강력해서 여러 유형의 프로덕션 환경에 모든 Python 모델을 빠르게 배포할 수 있습니다.

**MLflow는 여러 가지 일반적으로 사용하는 머신 러닝 라이브러리(예: scikit-learn, Spark MLlib, PyTorch, TensorFlow)에 대해 다양한 공통적 모델 “플레이버”가 미리 로드되어 있습니다.** 이런 모델 플레이버를 사용하면 모델을 처음에 생성한 이후에 손쉽게 기록하고 다시 로드할 수 있습니다. 이 **블로그 게시물**에서 예시를 확인하세요. 예를 들어 scikit-learn을 사용해서 MLflow를 사용하여 모델을 로깅할 때는 실험 내에서 다음 코드를 실행하기만 하면 됩니다.

```
mlflow.sklearn.log_model(model=sk_model, artifact_path=”sk_model_path”)
```

**MLflow는 “Python 함수” 플레이버도 제공합니다. 타사 라이브러리(예: XGBoost 또는 spaCy)의 모델이나 심지어 간단한 Python 함수 자체도 MLflow 모델로 저장할 수 있습니다.** Python 함수 플레이버를 사용하여 생성한 모든 Python 모델을 동일한 에코시스템에서 사용할 수 있고 추론 API를 통해 다른 MLflow 도구와 상호작용할 수 있습니다. 모든 사용 사례에 대해 계획하기는 불가능하지만, Python 함수 모델 플레이버는 최대한 보편적이고 유연하도록 설계되었습니다. 사용자 정의 처리와 로직 평가가 가능해서 ETL에 매우 유용합니다. 물론 더욱 “공식적인” 모델 플레이버가 온라인에 배포되고 있지만, 일반 Python 함수 플레이버도 여전히 “다양한 목적”으로 중요하게 사용됩니다. 모든 종류의 Python 코드와 MLflow의 로버스트 추적 툴킷을 연결해주는 역할을 합니다.

Python 함수 플레이버를 사용한 모델 로깅 과정은 간단합니다. **모델이나 함수를 모델로 저장할 때는 한 가지 요구 사항만 지키면 됩니다. pandas DataFrame을 입력값으로 받고 DataFrame 또는 NumPy 배열을 반환해야 합니다.** 이 요구 사항을 준수하고 나서 함수를 MLflow 모델로 저장할 때는 PythonModel에서 상속하는 Python 클래스를 정의하고, 사용자 정의 함수로 `.predict() 메서드`를 재정의합니다. 이 내용은 **여기**에 설명되어 있습니다.

## 런에서 로깅된 모델 로드

여러 신장 계수로 데이터를 실행해보았으므로 다음 단계에서는 결과를 검토하고 로깅된 지표를 기준으로 가장 성능이 좋은 모델을 찾아야 합니다. **MLflow를 사용하면 쉽게 로깅된 모델을 다시 로드하고 새로운 데이터에 대해 예측하는 데 사용할 수 있습니다. 이때 다음과 같은 명령을 사용합니다.**

1. 모델을 로드할 런의 링크를 클릭합니다.
2. '런 ID'를 복사합니다.
3. 모델이 저장된 폴더 이름을 기록합니다. 이 경우, "model"이라고 했습니다.
4. 모델 폴더 이름과 런 ID를 아래와 같이 입력합니다.

```
import custom_flavor as mlflow_custom_flavor

loaded_model = mlflow_custom_flavor.load_model(artifact_path='model', run_
id='e26961b25c4d4402a9a5a7a679fc8052')
```

모델이 원하는 대로 작동하는지 확인하기 위해 모델을 로드해서 `new_sales_units` 변수 내에서 생성한 새로운 제품 두 가지에 대한 DTW 거리를 측정합니다.

```
# use the model to evaluate new products found in 'new_sales_units'
output = loaded_model.predict(new_sales_units)
print(output)
```

## 다음 단계

보다시피 MLflow 모델이 손쉽게 처음 보는 새로운 값을 예측합니다. 추론 API를 따르기 때문에 어떤 서비스 플랫폼(예: Microsoft Azure ML 또는 Amazon SageMaker)에나 모델을 배포할 수 있으며, **로컬 REST API 엔드포인트**로 배포하거나 **사용자 정의 함수(UDF)**를 생성해 Spark-SQL에서 간편하게 사용할 수 있습니다. 지금까지 동적 시간 왜곡을 사용하여 Databricks Unified Data Analytics Platform에서 매출 동향을 예측하는 방법을 보여드렸습니다. 어서 **동적 시간 왜곡과 MLflow를 사용하여 매출 동향 예측** 노트북을 Databricks Runtime for Machine Learning을 사용해서 실행해보세요.

4장:

# 새로운 안전 재고 분석 전략으로 재고를 최적화하는 방법

글: Bryan Smith 및 Rob Saker

2020년 4월 22일

Databricks에서 이 노트북을 확인해보세요 →

어떤 제조기업은 고객의 주문을 처리하다가 공급업체에서 중요한 부품을 늦게 배송한다는 것을 발견합니다. 어떤 리테일러는 예상치 못한 이유로 인해 맥주 수요가 갑작스럽게 늘어났고, 공급이 부족해서 매출에 손해를 보기도 합니다. 고객은 기업 측의 공급 부족 때문에 부정적 경험을 합니다. 이런 기업들은 즉각적인 매출 기회를 놓치고 평판에 손상을 입습니다. 이런 상황이 익숙하게 들리시나요?

이상적인 세계에서는 상품 수요를 쉽게 예측할 수 있습니다. 현실에서는 아무리 예측을 잘해도 예상치 못한 사건에 영향을 받을 수밖에 없습니다. 원료 공급, 화물, 물류, 제조상의 고장, 예상치 못한 수요 상승 등으로 인해 혼란이 발생합니다. 리테일러, 유통업체, 제조업체, 공급업체 모두 이런 문제와 씨름하며, 재고를 과도하게 축적하지 않으면서도 고객의 수요에 안정적으로 대응하기 위해 노력합니다. 이때 새로운 안전 재고 분석 방법을 사용하면 도움이 될 수 있습니다.

기업에서는 필요한 곳에 리소스를 할당하여 예상 수요에 대응하기 위한 노력을 지속해서 기울입니다. 단기적 목표는 예측의 정확도를 개선하는 데 두는 경우가 많습니다. 이 목표를 달성하기 위해 확장할 수 있는 플랫폼, 사내 전문성 개발, 지능적인 새로운 모델에 투자합니다.

아무리 정밀한 예측도 미래를 완전히 맞출 수 없고, 수요가 갑작스럽게 바뀌면 재고가 동날 수 있습니다. 특히, 2020년 초에 코로나19 바이러스에 대한 우려가 치솟았을 때 **전국적으로 화장지 재고가 품귀 현상을 빚었습니다**. H-E-B 대표 Crag Boyan은 **이렇게 말했습니다**. “2개월 동안 판매할 분량을 단 2주 만에 판매했습니다.”

생산을 늘리는 단순한 방법으로는 문제를 해결할 수 없습니다. 유명한 화장지 제조사인 Georgia-Pacific의 **추산**에 따르면, 일반적 미국 가정은 팬데믹 이후 집에 머물게 되면서 화장지를 40% 더 많이 사용합니다. 이에 대응하여 Georgia-Pacific은 화장지를 생산하는 14개 시설에서 생산량을 20% 높였습니다. 생산량이 고정된 상태에서 대부분 장비가 하루 24시간, 7일 내내 가동되었기 때문에 생산량을 높이려면 추가 장비를 구매하거나 새로운 공장을 건설해서 생산 시설을 확대하는 수밖에 없습니다.

이렇게 생산량을 늘리면 업스트림 문제가 발생할 수 있습니다. 공급업체는 새로 확대, 확장된 제조 시설에서 요구하는 리소스를 제공하는 데 어려움을 겪을 수 있습니다. 화장지는 단순한 제품이지만, 미국, 캐나다, 스칸디나비아, 러시아의 산림 지역에서 펄프를 공급받고 지역적으로 재생 용지를 공급받아야 생산할 수 있습니다. 초기 비축분을 소진하고 나면 공급업체가 제조사에서 요구하는 재료를 수확해 가공하고, 배송하기까지 시간이 걸립니다.

이런 불확실성은 **채찍 효과**라는 공급망의 개념으로 표현할 수 있습니다. 공급망 전체에서 왜곡된 정보는 재고에 엄청난 비효율을 발생시키고, 화물과 물류비용을 높일 수 있으며, 생산 계획도 정확도가 하락하는 등의 문제를 일으킵니다. 제조업체나 리테일러가 재고를 정상으로 되돌리려고 하면 공급업체에서 생산량을 늘려야 하고, 업스트림 공급업체에서도 따라서 생산량을 늘려야 할 것입니다. 신중하게 관리하지 않으면 리테일러와 공급업체는 수요가 정상으로 돌아오거나, 소비자가 비축한 재고를 사용하느라 수요가 약간만 줄어들어도 과도한 재고와 생산 설비를 떠안게 됩니다. 이런 채찍 효과를 완화하려면 예측한 수요에 대한 불확실성을 철저히 검토하고 수요 역학을 **신중히 고려**해야 합니다.

## 안전 재고 분석으로 불확실성 관리

코로나19 팬데믹을 둘러싼 이런 소비자 수요 변화는 예측하기 어렵지만, 어떤 조직에서나 공급망을 관리하며 해결해야 하는 불확실성의 개념을 극단적인 예시로 보여줍니다. 비교적 정상적인 소비자 활동이 발생하는 기간에도 제품과 서비스에 대한 수요는 변동하고, 이를 고려하여 적극적으로 관리해야 합니다.


최신 수요 예측 도구는 수요에 대한 평균값을 예측하며, 주간 및 연간 계절적 효과, 장기 추세, 연휴 및 이벤트, 영향을 미치는 외부 요소(예: 날씨, 프로모션, 경제, 추가적 요소)를 고려합니다. 수요 예측에 한 개의 값만 산출하기 때문에 오해를 일으킬 수 있습니다. 절반 정도는 이 값보다 수요가 낮고 나머지 절반은 수요가 그보다 높게 나타나기 때문입니다.

평균 예측값을 이해하는 것은 중요하지만 양쪽의 불확실성을 이해하는 것도 그만큼 중요합니다. 이런 불확실성은 다양한 잠재적 수요 값을 제공하는 것으로 간주할 수 있으며, 각각의 수요 값에는 발생 가능성에 대한 수치화할 수 있는 확률이 있습니다. 예측에 대해 이런 식으로 생각하면 어떤 부분을 해결해야 할지 논의를 시작할 수 있습니다.

통계적인 관점에서 잠재적 수요의 전체 범위는 한계가 없으므로 100% 해결하기는 불가능합니다. 그러나 이론적 대화를 시작하기에 앞서 잠재적 수요 범위에 대응하는 능력을 증분적으로 개선할 때마다 재고 요구 사항이 상당히(실제로는 기하급수적으로) 증가한다는 것을 알게 되었습니다. 그래서 표적화된 **서비스 수준**을 겨냥해, 전체적 수요 범위에서 조직의 매출 목표와 재고 비용이 균형을 이루는 특정 부분을 해결하기로 했습니다.

이런 서비스 수준 기대를 설정할 경우, 평균 예측 수요를 해결하는 데 필요한 수량보다 일정 수준 추가 재고를 확보하여 불확실성에 대한 버퍼로 사용해야 합니다. 이런 안전 재고는 평균 정기 수요에 필요한 사이클 재고에 추가하면 전체적인 조직 목표가 균형을 이루면서도 (전부는 아니라도) 대부분 실제 수요 변동에 대응할 수 있게 됩니다.

## 필요한 안전 재고 수준 계산

기존 공급망 관련 문헌에서 안전 재고는 수요의 불확실성과 전달의 불확실성과 관련된 두 가지 공식 중 하나를 사용해서 계산합니다. 여기에서는 수요 불확실성을 다루기 때문에 불확실한 리드 타임은 고려하지 않아도 됩니다. 따라서, 하나의 단순화된 안전 재고 공식만 사용하면 됩니다.

$$\text{Safety Stock} = Z * \sqrt{PC/T} * \sigma_D$$

즉, 이 공식은 안전 재고가 평균 예측값($\sigma_D$)에 재고를 비축하는 (성과) 사이클 기간의 제곱근($\sqrt{PC/T}$)을 곱하고, 해당하는 범위 부분과 관련된 값(Z)을 곱한 수요의 평균 불확실성으로 계산된다는 것을 나타냅니다. 명확히 이해할 수 있도록 각 구성 요소에 대한 약간의 설명이 필요합니다.

앞의 섹션에서는 수요가 평균 값에 대한 잠재적 값의 범위로 존재한다고 설명했고, 우리 수요 예측에서 산출할 대상이기도 합니다. 이 범위가 평균에 대해 균등하게 분포되어 있다고 가정할 경우, 평균 값의 어느 한쪽에서 이 범위의 평균을 계산할 수 있습니다. 이를 표준 편차라고 합니다. σD 값은 수요의 표준 편차라고도 하는데, 평균에 대한 값 범위의 척도를 제공합니다.

이 범위가 평균을 중심으로 균형 있게 분포한다고 가정하였으므로 해당 평균의 표준 편차가 몇 개 존재하는 범위에 있는 값의 비율을 도출할 수 있습니다. 서비스 수준 기대를 사용하여 우리가 해결하고자 하는 잠재적 수요의 비율을 나타낼 경우, 안전 재고를 계획할 때 고려해야 할 수요의 표준 편차 개수를 살펴볼 수 있습니다. 값 범위의 비율을 알아내는 데 필요한 표준 편차 개수(z-점수, 공식에서 Z로 표현)를 산출하는 실제 계산이 다소 복잡해지지만 다행히 z-점수 표가 널리 공개되어 있고 **온라인 계산기**도 있습니다. 참고로 말씀드리면, 일반적으로 사용하는 서비스 수준 기대에 해당하는 z-점수 값은 다음과 같습니다.

| 서비스 수준 기대 | z (z-점수) |
|---|---|
| 80.00% | 0.8416 |
| 85.00% | 1.0364 |
| 90.00% | 1.2816 |
| 95.00% | 1.6449 |
| 97.00% | 1.8808 |
| 98.00% | 2.0537 |
| 99.00% | 2.3263 |
| 99.90% | 3.0902 |
| 99.99% | 3.7190 |

마지막으로 안전 재고를 계산하는 사이클 기간($\sqrt{}^{PC}/_{T}$)을 설명할 차례입니다. 제곱근 계산이 필요한 이유를 제쳐두고 남은 부분은 이 공식에서 가장 쉽게 이해할 수 있습니다. $^{PC}/_{T}$ 값은 안전 재고를 계산하려는 사이클 기간을 나타냅니다. T로 나누는 이유는 이 기간을 표준 편차 값을 계산하는 데 사용한 것과 동일한 단위로 표현해야 하기 때문일 뿐입니다. 예를 들어, 안전 재고를 7일 사이클로 준비하려는 경우에는 이 기간에 대해 7의 제곱근을 취할 수 있습니다. 단, 일일 수요 값을 사용하여 수요의 표준 편차를 계산한 상태여야 합니다.

## 추정이 까다로운 수요 분산

표면적으로 안전 재고 분석 요구 사항의 계산은 상당히 간단합니다. 공급망 관리 수업에서는 학생들에게 수요에 대한 과거 값을 제공하는 경우가 많습니다. 학생들은 공식의 표준 편차 부분을 계산합니다. 서비스 수준 기대가 있다면, z-점수를 빠르게 도출하고 이 목표 수준에 도달하기 위한 안전 재고 요구 사항을 알아낼 수 있습니다. 하지만 이 수치는 잘못됐습니다. 좀 더 정확히 말하면 중요한 가정을 벗어나면 이 수치는 맞지 않는데, 그 중요한 가정은 거의 절대로 유효한 법이 없습니다.

모든 안전 재고 계산의 난제는 수요의 표준 편차를 구하는 것입니다. 표준 공식은 계획하는 미래의 기간에서 수요와 관련된 변동을 알아야 도출할 수 있습니다. 시계열의 변동이 안정적인 경우는 극히 드뭅니다. 데이터의 추세와 계절적 패턴에 따라 변동하는 경우가 많습니다. 이벤트와 외부 설명 변수도 영향을 미칩니다.

이 문제를 극복하기 위해 공급망 소프트웨어 패키지는 예측 오류 척도(예: 오차의 제곱 평균 제곱근(RMSE), 절대 오차 평균(MAE))을 수요의 표준 편차로 대체하는 경우가 대부분이지만 이 값은 (관련 개념이기는 하지만) 다른 것을 나타냅니다. 그래서 안전 재고 요구 사항이 과소하게 추정되는 경우가 많습니다. 이 그래프에서 95% 기대를 설정했음에도 불구하고 서비스 수준은 92.7%를 달성한 것을 참고하세요.


대부분 예측 모델은 예측 평균을 계산하면서도 오류를 최소화하려고 애쓰지만, 역설적으로 모델 성능을 개선하면 오히려 과소 추정 문제가 악화된다는 경우가 많습니다. 아마 그래서 많은 리테일러가 공개된 서비스 수준 기대를 달성하려고 노력하지만 대부분이 실패하고 있다는 **인식이 커지고 있는 듯합니다**.

## 그렇다면 이제 어떻게 해야 하고, Databricks는 어떤 도움을 줄 수 있을까요?

이 문제를 해결하는 중요한 첫 단계는 안전 재고 분석 계산의 단점을 인식하는 것입니다. 하지만 인정하는 것만으로는 부족한 경우가 많습니다.

일부 연구자들이 안전 재고 추정을 개선하겠다는 명확한 목적을 가지고 수요 분산 추정의 정확도를 높이는 **기술**을 정의하려고 하지만, 그 방법에 대해서는 의견이 분분합니다. 이 기술을 편리하게 구현할 수 있는 소프트웨어도 쉽게 구할 수 없습니다.

지금은 공급망 관리자 여러분께서 과거 서비스 수준 성과를 신중하게 검토하고 목표하는 타겟을 달성하고 있는지 확인해보시는 것이 좋습니다. 그러려면 과거 예측 데이터와 과거 실제 데이터를 신중하게 결합해야 합니다. 기존 데이터베이스 플랫폼에 데이터를 저장하는 비용으로 인해 많은 조직에서 과거 예측이나 매우 세부적인 소스 데이터를 보관하지 않지만, 클라우드 기반 스토리지를 사용하고 온디맨드 연산 기술을 통해 압축된 고성능 형식으로 저장된 데이터에 액세스한다면(Databricks 등의 플랫폼 경유) 비용 효율적으로 계산하고 많은 조직에서 쿼리 성능을 개선할 수 있습니다.

자동화되거나 디지털 지원을 받는 주문 처리 시스템(많은 사람이 온라인에서 구매하고 매장에서 수령하는(BOPIS) 모델에 필요)을 배포하고 주문 처리에 대한 실시간 데이터를 생성하기 시작하면서 기업에서는 이 데이터를 사용하여 품절 문제를 탐지합니다. 이는 서비스 수준 기대와 매장 내 재고 관리 현황을 다시 평가할 필요가 있음을 나타냅니다. 일상적 운영에서 이러한 분석을 실행하는 데 제한이 있었던 제조업체는 교대 근무조에 따라 분석하고 조정하고 싶을 수 있습니다. Databricks의 스트리밍 입력 기능은 기업에서 거의 실시간에 가까운 데이터로 안전 재고 분석을 수행하도록 지원할 수 있는 해법을 제공합니다.

마지막으로 재고 계획 프로세스에 더 나은 입력을 제공할 수 있는 새로운 예측 방법을 고려하는 것이 좋습니다. Facebook Prophet과 더불어, Databricks와 같은 병렬화 및 자동 확장 플랫폼을 사용한 여러 기업에서 시기적절하고 세분화된 예측이 가능했습니다. **GARCH(Generalized Autoregressive Conditional Heteroskedastic)** 모델 등의 다른 예측 기술을 사용하면 수요 변동을 검토하여, 안전 재고 전략을 설계하는 데 매우 유용할 수 있습니다.

안전 재고를 비축하고자 하는 기업은 이 문제를 해결하면 상당한 잠재적 이익이 있지만 최종 상태로 가는 길이 명확하지 않기 때문에 유연성이 성공의 핵심이 될 것입니다. Databricks는 이러한 여정을 돕는 도구로써 확실한 경쟁력이 있다고 생각합니다. 우리 고객들과 이 길을 함께 헤쳐 나가고 싶습니다.

*이 중요한 주제에 인사이트를 주신 Southern Methodist University Cox School of Business 의 **Sreekumar Bhaskaran** 교수에게 감사드립니다.*

> 이 무료 Databricks **노트북**으로 실험을 시작하세요.

5장:

# 공급망 수요 예측을 개선하기 위한 새로운 방법

인과적 요소로 세분화된 수요 예측

글: Bryan Smith 및 Rob Saker

2020년 3월 26일

## 기업에서 세분화된 수요 예측 도입 가속화

리테일러와 소비자 상품 제조업체들은 비용을 낮추고, 운영 자본을 해방하고, 옴니채널 혁신의 기반을 마련하기 위해 공급망 관리를 개선하기 위한 노력을 강화하고 있습니다. 소비자 구매 행동의 변화는 공급망에 새로운 압력을 가하고 있습니다. 수요 예측을 통해 소비자 수요에 대한 이해를 높이는 것은 이런 노력을 시작하는 좋은 시작점으로 여겨집니다. 제품과 서비스에 대한 수요를 알면 인력, 재고 관리, 공급 및 생산 계획, 화물, 물류 등 여러 가지를 결정할 수 있기 때문입니다.

McKinsey & Company의 ***Notes from the AI Frontier***에서는 리테일 공급망 예측의 정확도를 10~20% 높이면 재고 비용이 5% 감소하고 매출이 2~3% 늘어날 것이라고 언급했습니다. 기존 공급망 예측 도구는 이런 바람직한 결과를 제공하지 못합니다. 리테일 공급망 수요 예측에서 **산업 평균 부정확도는 32%**이기 때문에 예측이 아주 약간만 개선되어도 대부분 리테일러에 엄청난 차이로 다가올 것입니다. 그러므로 많은 기업이 사전에 패키지로 구성된 예측 솔루션에서 벗어나, 내부적으로 수요 예측 기술을 키울 방법을 모색하고 연산 효율을 높이는 대신 예측의 정확도를 저하시키던 기존의 방식을 다시 검토하고 있습니다.

이러한 노력의 핵심은 시간 및 (위치/제품) 계층을 더욱 세분화하여 예측하는 데 있습니다. 세분화된 수요 예측을 사용하면 필수적인 수준에 가까운 수요에 영향을 미치는 패턴을 찾아낼 수도 있습니다. 예전에는 리테일러가 시장 또는 유통 수준에서 어떤 제품 분류에 대해 최대 1개월 또는 일주일의 단기 수요를 예측한 다음, 그 예측값을 사용해서 해당 분류의 제품 단위를 어떻게 특정 매장과 요일에 배치할지 결정했습니다. 세분화된 수요 예측은 특정 위치에 있는 특정 제품의 역학을 반영하는 더욱 지역화된 모델을 구축할 수 있습니다.

## 세분화된 수요 예측의 문제

세분화된 수요 예측이 유망하기는 하지만 여러 가지 어려움이 있습니다. 먼저 집계 예측에서 벗어나게 되면 생성해야 할 예측 모델과 예측 수가 폭발적으로 늘어납니다. 기존 예측 도구로는 필요한 처리 수준을 달성하기 어렵거나, 이 정보를 유용하게 사용할 수 있는 서비스 기간을 훨씬 넘기기 마련입니다. 이런 제약으로 인해 기업들은 분석에서 처리하는 카테고리 수나 세분화 수준을 제한할 수밖에 없습니다.

이전 **블로그 게시물**에서 살펴보았듯이, Apache Spark를 사용하여 이러한 문제를 극복하고 시기적절하고 효율적으로 작업을 병렬화해서 실행할 수 있습니다. Databricks와 같은 클라우드 네이티브 플랫폼에 배포하면 컴퓨팅 리소스를 신속히 할당했다가 거두어들일 수 있어서 작업 비용을 예산 내로 통제할 수 있습니다.

두 번째 문제이자 좀 더 극복이 어려운 문제가 있다면, 데이터를 더욱 세분화해서 검토할 때 집계된 형태의 수요 패턴이 존재하지 않을 수 있다는 것입니다. 아리스토텔레스의 말을 빌리자면, 전체가 부분의 합보다 큰 경우가 많습니다. 분석을 점점 세분화할수록, 간략한 수준에서 쉽게 모델링되었던 패턴이 없을 수도 있고 간략한 수준에서 적용할 수 있는 기술로 예측하기가 더욱 어려워집니다. 예측의 이러한 문제는 1950년대의 **Henri Theil**를 비롯한 수많은 실무자가 언급한 바 있습니다.

세분화가 트랜잭션 레벨에 가까워지면 각 고객 수요와 구매 결정에 영향을 미치는 외부의 인과적 요소를 고려해야 합니다. 집계 수준에서는 시계열을 구성하는 평균, 추세, 계절성에 반영되지만 분석을 세분화하면 이러한 요소를 예측 모델에 직접 포함해야 할 수도 있습니다.

마지막으로 분석을 세분화하면 데이터 구조상 기존 예측 기술을 사용하지 못하게 될 가능성도 커집니다. 트랜잭션 수준에 가까워질수록 데이터에서 비활성 기간을 다루어야 할 가능성도 커집니다. 이런 세분화 수준에서는 판매 개수 등의 수량 데이터를 처리할 때 특히 종속적 변수가 단순한 변환만으로는 처리할 수 없을 정도로 분포를 왜곡시켜서, 대부분 데이터 사이언티스트가 익숙하게 사용하던 것 외의 다른 예측 기술을 사용해야 할 수도 있습니다.

## 과거 데이터 액세스

**자세한 내용은 데이터 준비 노트북을 참조하세요. →**

이런 문제를 해결하기 위해 뉴욕시 자전거 공유 프로그램(Citi Bike NYC)의 공개 이동 기록 데이터를 활용할 것입니다. Citi Bike NYC는 시민들이 '자전거를 발견하고, 뉴욕을 발견하도록' 돕겠다는 약속을 내걸었습니다. 이 서비스를 사용하는 시민들은 뉴욕시 전역의 850개 이상 대여 매장에 가서 자전거를 대여할 수 있습니다. 자전거를 13,000대 이상 보유하고 있고 40,000대로 늘릴 계획입니다. 가입자 100,000명 이상이 매일 자전거로 약 14,000회를 달립니다.

Citi Bike NYC는 반납된 곳에서 향후 수요가 예측되는 곳으로 자전거를 이동합니다. Citi Bike NYC는 리테일러와 소비자 상품 기업이 일상에서 씨름하는 문제와 비슷한 어려움이 있습니다. 어떻게 하면 수요를 가장 잘 예측하여 적절한 지역에 리소스를 할당할 수 있을지 고민합니다. 수요를 과소 추정하면 수익을 낼 기회를 놓치고 고객이 불만을 느낄 수도 있습니다. 과하게 추정하면 사용하지 않는 자전거 재고가 너무 많아집니다.

이 공개 제공되는 데이터 세트는 전월 말부터 2013년 중반에 프로그램이 도입된 시점까지 거슬러 올라가 각 자전거 임대에 대한 정보를 제공합니다. 이동 기록 데이터는 특정 대여점에서 자전거를 대여한 정확한 시간과 다른 대여점에 자전거가 반환된 시간이 나와 있습니다. Citi Bike NYC 프로그램의 대여점을 매장 위치로 생각하고, 대여를 시작한 것을 거래로 생각한다면, 이는 예측할 수 있는 장기적이고 상세한 거래 기록에 가까운 데이터라고 할 수 있습니다.

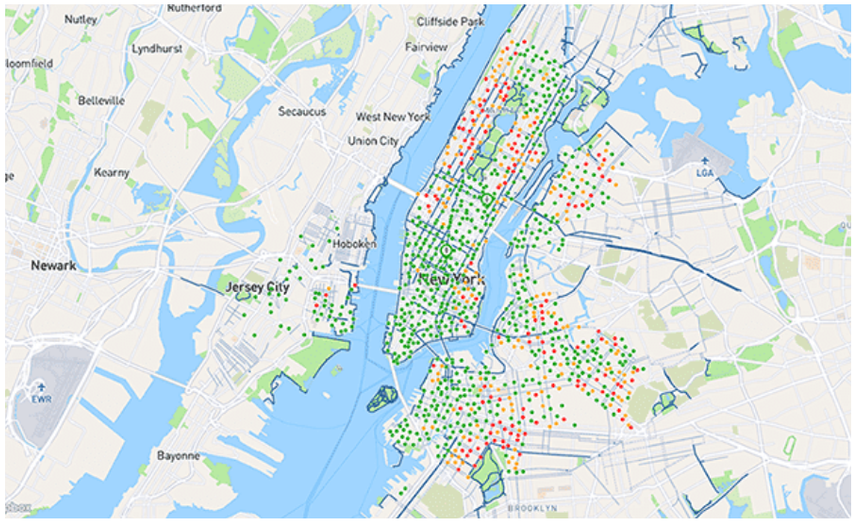
이 예시에서는 모델링에 포함할 외부 요소를 찾아야 합니다. 연휴 이벤트와 과거(및 예측된) 기상 데이터를 외부 영향 요소로 사용할 것입니다. 연휴 데이터 세트의 경우, Python에서 **holidays 라이브러리**를 사용하여 2013년부터 현재까지 표준 연휴를 찾습니다. 날씨 데이터의 경우, 흔히 사용하는 날씨 데이터 집계 도구인 **Visual Crossing**에서 시간당 날씨를 추출할 것입니다.

Citi Bike NYC와 Visual Crossing 데이터 세트는 이용 약관상 데이터를 직접 공유할 수 없습니다. 우리가 예측한 결과를 재현해보고 싶은 분은 데이터 제공자의 웹사이트에서 이용 약관을 검토하고, 데이터 세트를 환경에 적절히 다운로드하세요. 이런 원시 데이터 자산을 분석에 사용된 데이터 개체로 변환하는 데 필요한 데이터 준비 로직은 저희가 제공합니다.

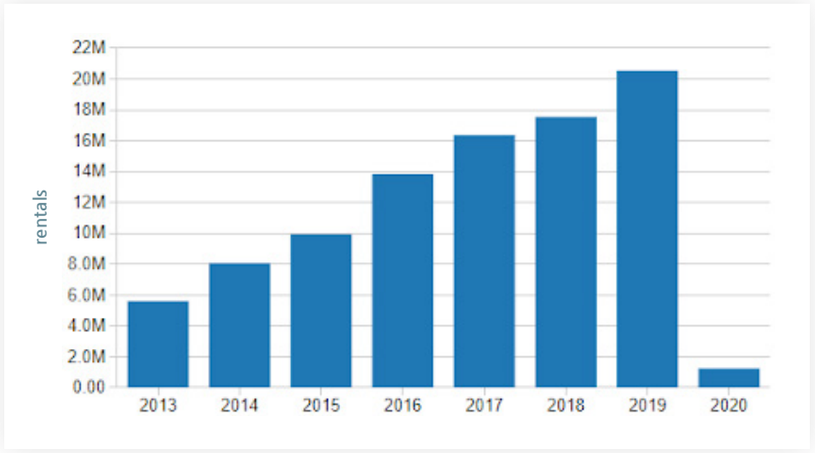
## 거래 데이터 검토

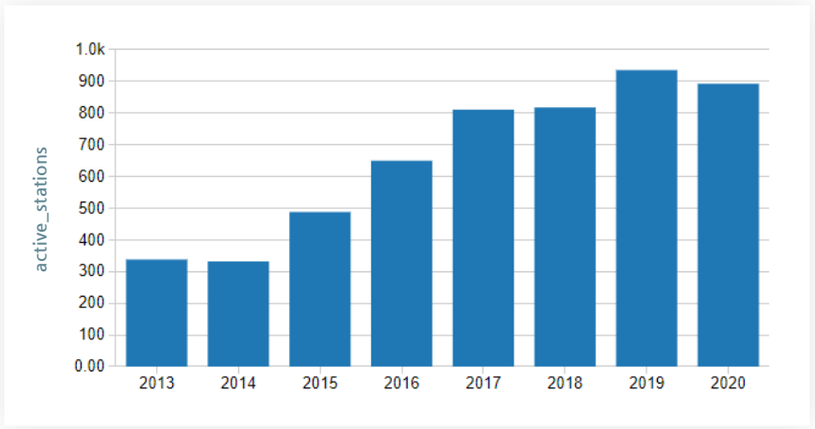
**자세한 내용은 탐색적 분석 노트북을 참조하세요. →**

2020년 1월 기준으로 Citi Bike NYC 자전거 공유 프로그램은 뉴욕시 대도시권, 주로 맨해튼 지역에 대여점을 864곳 운영하고 있습니다. 2019년에만 고객이 시작한 고유 대여 건수가 400만을 넘겼고, 대여가 많은 날에는 약 14,000건이 발생했습니다.

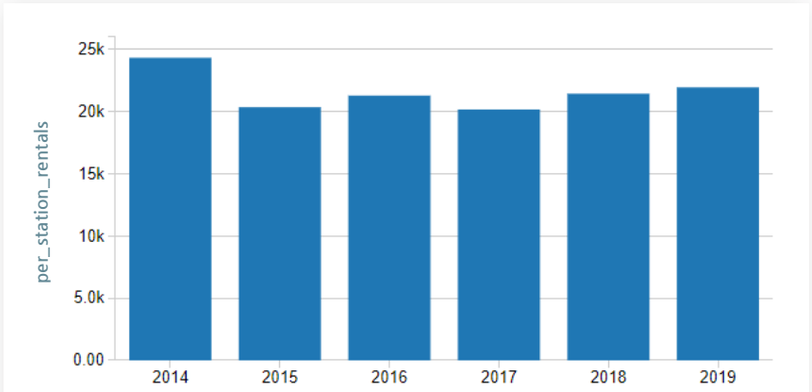



이 프로그램을 시작한 이후로 매년 대여 건수가 증가했습니다. 이렇게 대여 건수가 증가한 것은 자전거 사용률이 늘어난 탓이겠지만 전반적인 대여점 네트워크가 확장과도 대체로 일치합니다.

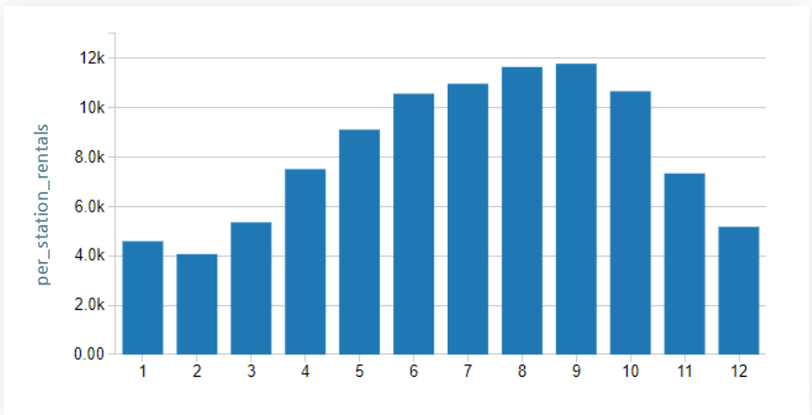


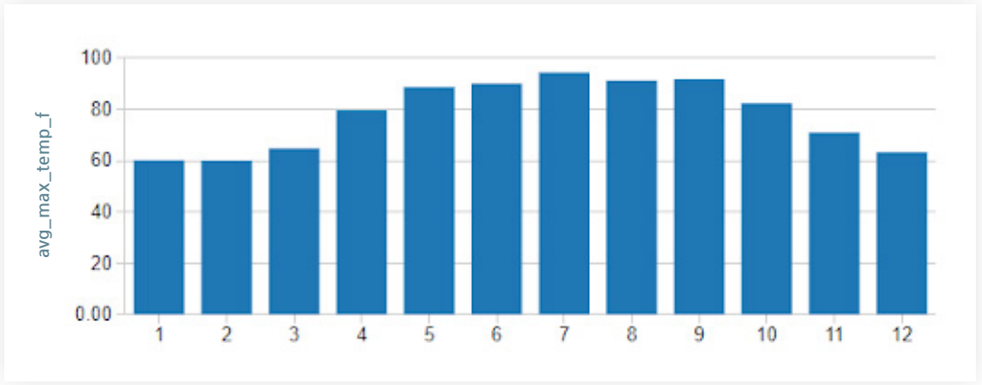
네트워크에서 운영 중인 대여점 수를 기준으로 대여 건수를 정규화했을 때 최근 몇 년 동안 대여점별 자전거 이용자 수가 서서히 늘어서 약간의 선형 상향 추세를 형성했습니다.



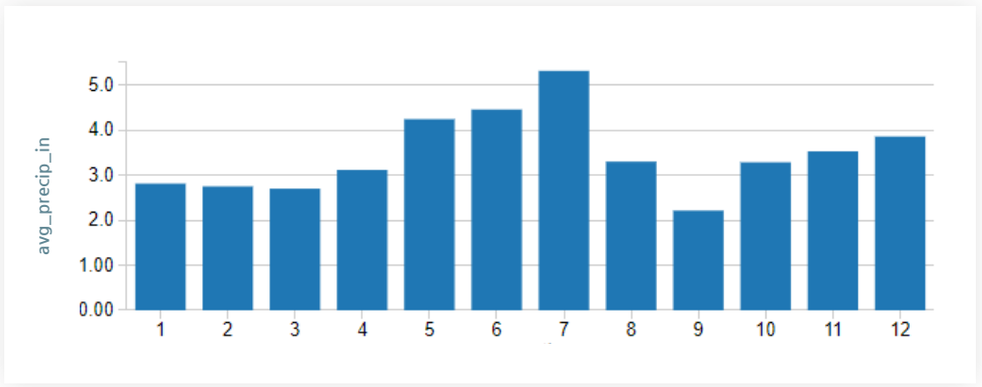

이 정규화된 값을 대여 건수에 적용했을 때, 자전거 이용자 수에서 독특한 계절적 패턴이 나타났습니다. 봄, 여름, 가을에는 늘었다가 자전거를 타기에는 바깥 날씨가 좋지 않은 겨울에는 하락했습니다.



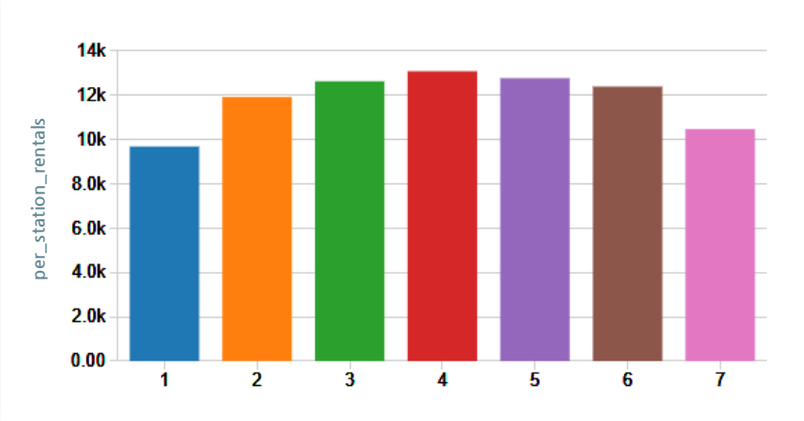

이 패턴은 뉴욕시의 최대 기온(화씨) 패턴을 거의 근접하게 따르는 듯합니다.



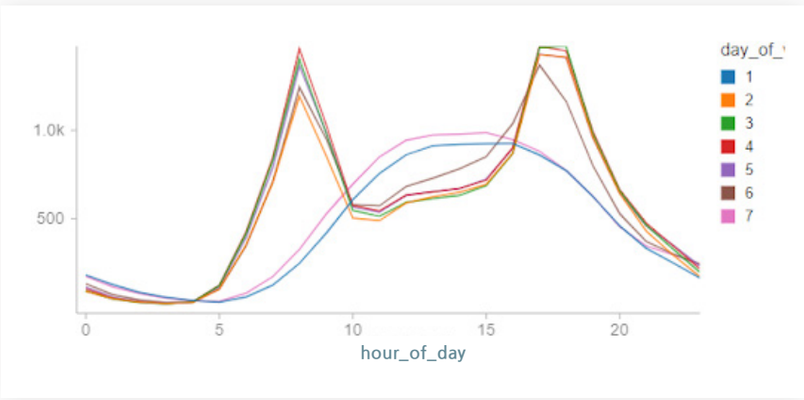

월간 이용자 수와 기온 패턴은 거의 유사하지만 강우량(월평균 인치)은 이러한 패턴을 그다지 따라가지 않았습니다.



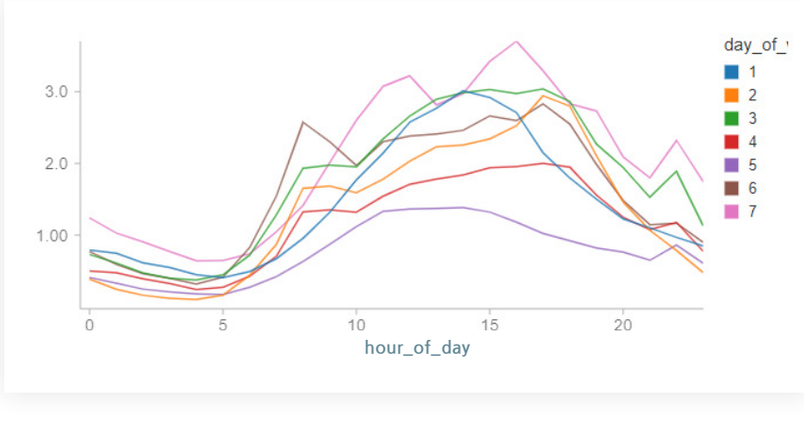
일요일의 주간 이용자 수 패턴은 1이고, 토요일은 7인 것으로 보아 뉴욕 시민들은 자전거를 출퇴근 수단으로 이용하는 듯합니다. 이는 다른 여러 자전거 공유 프로그램에서도 나타난 패턴입니다.



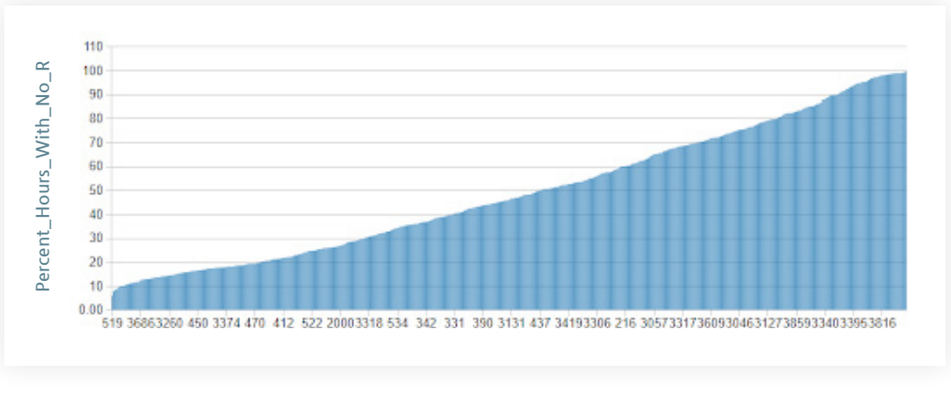

이런 이용자 수 패턴을 시간별로 나누면 주중에 이용자 수가 표준 통근 시간에 급증하는 패턴을 확인할 수 있습니다. 주말에는 프로그램을 여가용으로 활용하는 패턴이 나타나, 앞서 세운 가설을 뒷받침했습니다.



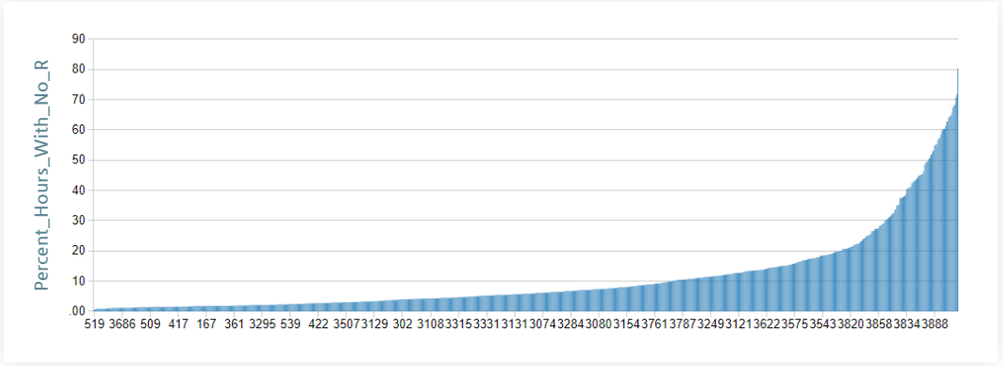

흥미로운 패턴이 있다면 요일과 관계없이 연휴에는 주말 사용 패턴과 거의 유사한 사용 패턴이 나타났습니다. 연휴가 수시로 있기 때문에 추세가 불규칙하게 나타날 수도 있습니다. 하지만 그래프로 보았을 때는 연휴를 구분하는 것이 신뢰할 수 있는 예측을 하는 데 중요한 듯합니다.




시간당 집계 결과에서는 뉴욕시가 정말로 불야성인 것처럼 보입니다. 실제로는 자전거 대여가 전혀 없는 시간이 긴 대여점이 많이 있습니다.



이런 활동의 격차는 예측 시 문제의 소지가 있습니다. 1시간에서 4시간 간격으로 이동하면 각 대여점에서 대여 활동이 일어나지 않는 시간이 상당히 감소하지만 여전히 그 시간에 활동이 일어나지 않는 대여점이 많습니다.




세분화 수준을 낮춰서 활동이 없는 기간의 문제를 숨기기보다는 시간당 수준에서 예측하기로 하고, 이 데이터를 처리하는 데 도움이 될 만한 대체 예측 기술을 찾아보기로 했습니다. 활동이 거의 없는 대여점에 대한 예측은 흥미로운 부분이 없어 활동이 가장 많은 상위 200개 대여점으로만 분석을 제한하도록 하겠습니다.

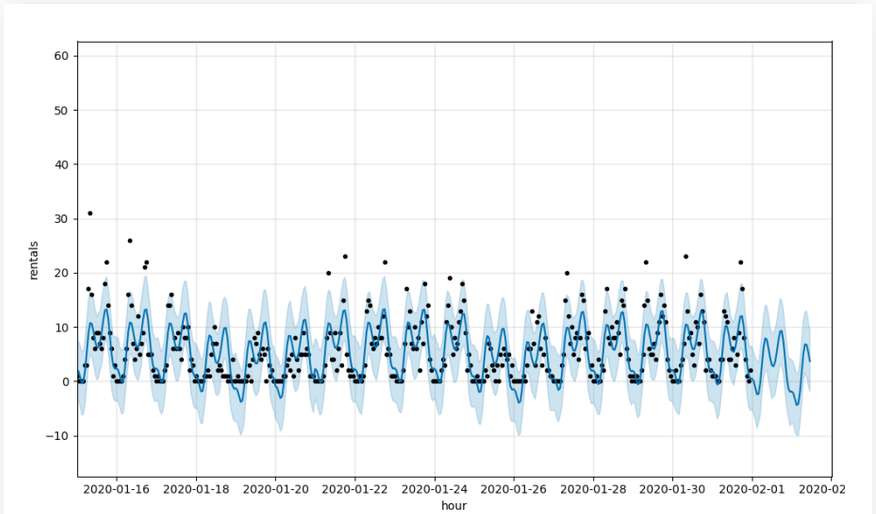
## Facebook Prophet을 사용한 자전거 공유 대여 예측

대여점 수준에서 자전거 대여를 예측하기 시작했을 때 시계열 예측에 흔히 사용하는 Python 라이브러리인 Facebook Prophet을 사용했습니다. 이 모델은 일일, 주간, 연간 계절적 패턴과 선형 성장 패턴을 탐색하도록 구성되었습니다. 연휴와 관련된 데이터 세트의 기간도 구분해서 이 날짜의 비정상적인 활동이 알고리즘에서 탐지된 평균, 추세, 계절적 패턴에 영향을 받지 않도록 했습니다.

앞서 언급한 블로그 게시물에 나와 있는 확장 패턴을 사용하여 가장 활동이 많은 대여점 200개에 대해 모델을 훈련하였고 각각에 대해 36시간 예측을 생성했습니다. 모두 합쳐서 모델은 평균 제곱근 오차(RMSE)가 5.44이고, 중간 평균 비례 오차(MAPE)는 0.73이었습니다. (실제 0값은 MAPE 계산 시 1로 보정했습니다.)

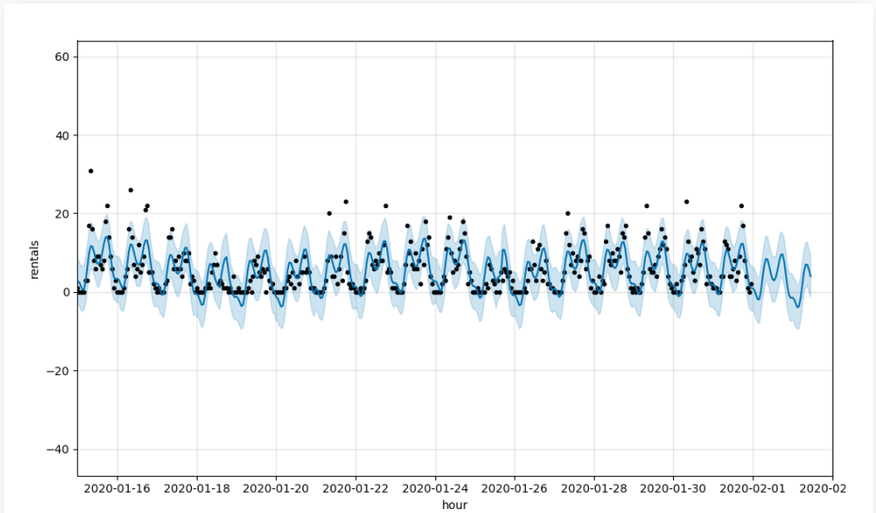
이 지표는 모델들이 대체로 대여에 대해서는 잘 예측했지만, 시간당 대여율이 높은 시점은 제공하지 못했습니다. 각 대여점의 매출 데이터를 시각화하면 그래프로 확인할 수 있습니다. E 39 St 및 2 Ave에 있는 대여점 518은 RMSE가 4.58이고 MAPE가 0.69입니다.

**자세한 내용은 시계열 노트북을 참조하세요. →**




그런 다음, 이 모델에 기온과 강수량을 설명 변수로 포함했습니다. 전체 예측 결과는 RMSE가 5.35이고 MAPE가 0.72였습니다. 모델이 다소 개선되기는 했지만 여전히 대여점에서 이용자 수가 큰 폭으로 변화하는 양상은 찾아내지 못했습니다. 아까와 마찬가지로 대여점 518(RMSE: 4.51, MAPE: 0.68)을 보면 알 수 있습니다.

**자세한 내용은 설명 변수를 포함한 시계열 노트북을 참조하세요 →**




이렇게 두 시계열 모델에서 큰 값을 모델링하는 데 어려움을 겪는 것은 **Poisson 분포**가 있는 데이터를 처리할 때 **일반적**으로 나타나는 패턴입니다. 이런 분포에서는 평균 주변에 많은 값이 몰려 있고 평균 위로는 롱테일이 나타납니다. 평균 반대쪽은 바닥이 0이어서 데이터가 왜곡됩니다. 현재 Facebook Prophet에서는 데이터가 정규 (Gaussian) 분포를 가져야 하지만 Poisson 회귀를 포함하는 **계획**이 논의되고 있습니다.

## 공급망 수요 예측을 위한 대안

그러면 이 데이터는 어떻게 예측해야 할까요? Facebook Prophet 관리자들이 계획하고 있는 것처럼, 기존 시계열 모델에서 Poisson 회귀 기능을 제공하는 방법이 있습니다. 이는 좋은 수단일 수 있지만 자료가 많지 않기 때문에 다른 기술을 고려하기도 전에 이 방법을 시도하는 것은 적절하지 않을 수 있습니다.

또 다른 해결 방법은 0이 아닌 값의 지표와 값이 0인 기간이 발생하는 빈도를 모델링하는 것입니다. 그런 다음, 각 모델의 결과를 결합해 예측을 생성합니다. 이는 크로스톤 방법(Croston's method)이라고 하는데, 최근에 출시된 크로스톤 Python 라이브러리에서 지원되기는 하지만 어떤 데이터 사이언티스트들은 직접 함수를 구현하기도 했습니다. 그러나 이 방법은 (1970년대에 나왔지만) 흔히 사용하지 않기 때문에 저희는 좀 더 기본적인 방법을 찾아보려고 합니다.

이런 이유로 랜덤 포레스트 회귀 모델이 적절할 듯합니다. 일반적으로 결정 트리는 다른 여러 통계법과 달리 데이터에 동일한 제약을 적용하지 않습니다. 예측 변수에 대한 값 범위를 보면 제곱근 변환 등의 방법을 사용해서 대여 건수를 변환하고 모델을 훈련해야 할 듯합니다. 그렇게 하지 않더라도 알고리즘의 성능 정도는 확인할 수 있을 것입니다.

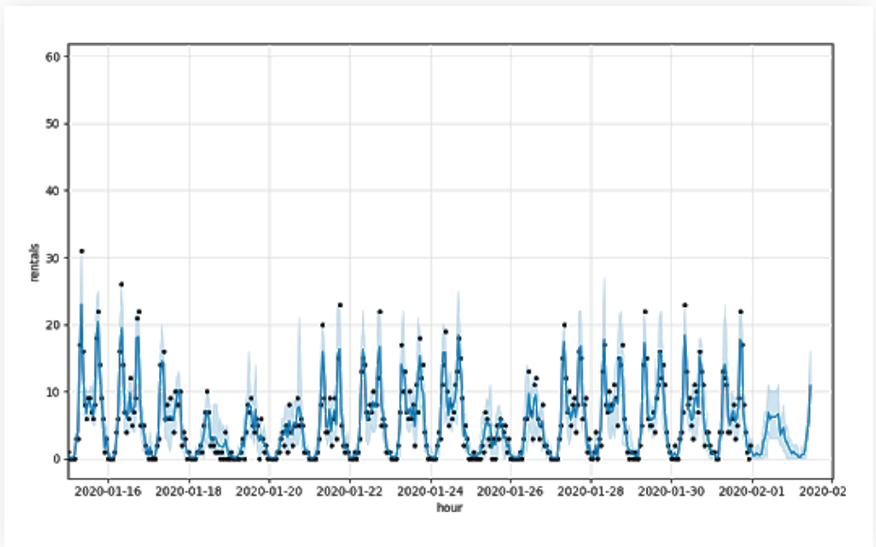
이 모델을 사용하려면 몇 가지 feature를 설계해야 합니다. 탐색적 분석에서 연간, 주간, 일일 단위로 데이터에 명확한 계절적 패턴이 있다는 것을 확실히 알게 되었습니다. 그래서 연간, 월간, 일일, 시간을 feature로 추출했습니다. 연휴에 적용할 플래그를 포함할 수도 있습니다.

랜덤 포레스트 회귀자와 시간에서 파생된 feature만 사용했을 때 전체 RMSE는 3.4이고 MAPE는 0.39였습니다. 대여점 518의 경우, RMSE와 MAPE 값은 각각 3.09와 0.38이었습니다.

**자세한 내용은 시간 노트북을 참조하세요. →**

이런 동일한 시간 feature와 결합한 강우량 및 기온 데이터를 활용한 덕분에 높은 대여 값 중 일부는 (완벽히는 아니라도) 좀 더 잘 해석할 수 있었습니다. 대여점 518의 RMSE는 2.14로 하락했고, MAPE는 0.26으로 하락했습니다. 전체 RMSE는 2.37로, MAPE는 0.26으로 하락해서 날씨 데이터가 자전거 수요 예측에 유용하다는 것을 알 수 있습니다.

**자세한 내용은 시간 및 날씨 feature를 적용한 랜덤 포레스트 노트북을 참조하세요. →**



## 결과의 의미

수요를 세분화해서 예측하려면 모델링 방법에 대한 생각을 바꿀 필요가 있습니다. 간략한 시계열 패턴에서는 요약하더라도 문제가 없는 외부 영향 요소를 모델에 더욱 명시적으로 포함해야 합니다. 집계 수준에서는 데이터 분포에 숨겨지는 패턴이 더욱 명확하게 노출되고, 따라서 모델링 방식을 바꿔야 할 수도 있습니다. 이 데이터 세트에서는 시간당 날씨 데이터를 포함하여 이 문제를 해결하였고 기존 시계열 기술 대신 입력 데이터에 대한 가정이 적은 알고리즘으로 바꾸었습니다.

그 외에도 살펴볼 만한 외부 영향 요소와 알고리즘이 많이 있고, 이를 시험하는 과정에서 유난히 데이터 하위 집합에 효과적인 것을 발견할 수도 있습니다. 또한, 새로운 데이터가 생기면 이전에는 적용했던 방법을 포기하고 새로운 기술을 고려해야 할 수도 있습니다.

세분화된 수요 예측을 탐색하는 고객들은 각 훈련 및 예측 사이클에서 여러 가지 기술을 평가하는 패턴을 공통으로 보였습니다. 이를 '자동 모델 베이크오프'라고 합니다. 베이크오프 라운드에서 특정 데이터 하위 집합에 최상의 결과를 제공하는 모델을 선택하고, 각 하위 집합에서 효과적인 모델 타입을 결정할 수 있습니다. 최종적으로는 데이터와 적용하는 알고리즘이 적절히 일치하는 우수한 데이터 사이언스를 활용하고 싶지만, 그동안 여러 문서에서도 말씀드렸듯이 문제의 정답은 하나가 아니며 어떤 시점에서 특히 효과적인 솔루션이 있을 수도 있습니다. Apache Spark, Databricks 등의 플랫폼에서는 컴퓨팅 기능에 액세스하여 이런 모든 경로를 둘러보고 비즈니스에 맞는 최적의 솔루션을 제공할 수 있습니다.

## 추가적 리테일/CPG 및 수요 예측 참고 자료

실험을 시작할 수 있는 개발자 리소스:

1. 노트북:
   - **데이터 준비 노트북**
   - **탐색적 분석 노트북**
   - **시계열 노트북**
   - **설명 변수를 포함한 시계열 노트북**
   - **시간 노트북**
   - **시간 및 날씨 feature를 포함한 랜덤 포레스트 노트북**
2. **리테일 및 CPG용 대규모 데이터 분석 및 AI 가이드**를 다운로드하세요.
3. **리테일 및 CPG** 페이지에서 Dollar Shave Club과 Zalando가 Databricks로 혁신한 사례를 알아보세요.
4. 최신 블로그 **Facebook Prophet 및 Apache Spark를 사용한 세분화된 대규모 시계열 예측**을 읽고 **Databricks Unified Data Analytics Platform**에서 어떻게 시기적절하고 기업에서 제품 재고를 정밀하게 제공할 수 있는 세분화 수준으로 문제를 해결하는지 알아보세요.

6장:

# Facebook Prophet 및 Apache Spark™를 사용한 세분화된 대규모 시계열 예측

글: Bilal Obeidat, Bryan Smith 및 Brenner Heintz

2020년 1월 27일

Databricks에서 이 시계열 예측 노트북을 확인해보세요 →

시계열 예측 기술이 발전한 덕분에 리테일러는 더욱 신뢰할 수 있는 수요를 예측할 수 있게 되었습니다. 지금은 기업에서 정밀하게 제품 재고를 조정할 수 있을 정도로 이런 예측을 시기적절하고 세분화하여 수행해야 한다는 문제가 남아 있습니다. 이 문제에 직면한 기업들은 Apache Spark 및 Facebook Prophet을 사용하면서 과거의 솔루션에 존재하던 확장성과 정확도의 한계를 극복하고 있습니다.

이 게시물에서는 시계열 예측의 중요성을 설명하고, 몇 가지 샘플 시계열 데이터를 시각화한 다음, 간단한 모델을 구축해 Facebook Prophet으로 보여드릴 것입니다. 모델 하나를 구축하는 방법을 알고 나면 Prophet과 마법과도 같은 Apache Spark™를 결합하여 한 번에 수백 개의 모델을 훈련해보겠습니다. 이를 통해 개별 제품-매장 조합에 대해 지금까지는 보기 어려웠던 수준으로 세분화된 정밀한 예측을 예측할 수 있습니다.

## 정확하고 시기적절한 예측이 그 어느 때보다 중요

시계열 분석의 속도와 정확도를 개선하여 제품과 서비스의 수요 예측을 개선하는 것은 리테일러의 성공에 매우 중요합니다. 지나치게 많은 제품을 매장에 배치할 경우, 선반과 창고 공간이 부족해지고 제품 유통 기한이 지날 수 있습니다. 리테일러들은 재고에 자금이 묶여서 제조사나 소비자 패턴 변화에서 발생하는 새로운 기회를 놓칠 수도 있습니다. 매장에 제품이 너무 적으면 고객들은 필요한 제품을 구매하지 못할 수 있습니다. 이런 예측 오류는 리테일러에 수익 손실을 바로 입힐 뿐만 아니라, 장기적으로도 소비자 불만이 커져 경쟁사로 소비자가 이탈할 수도 있습니다.

**새로운 기대로 인해 더욱 정밀한 시계열 예측 방법과 모델이 필요**

한동안 기업 리소스 계획(ERP) 시스템과 타사 솔루션은 간단한 시계열 모델을 기반으로 수요 예측 기능을 제공했습니다. 하지만 기술이 발전하고 이 부문에서의 압력이 커지면서 많은 리테일러가 기존의 선형 모델과 더욱 전통적인 알고리즘에서 벗어날 방법을 모색하고 있습니다.

PROPHET

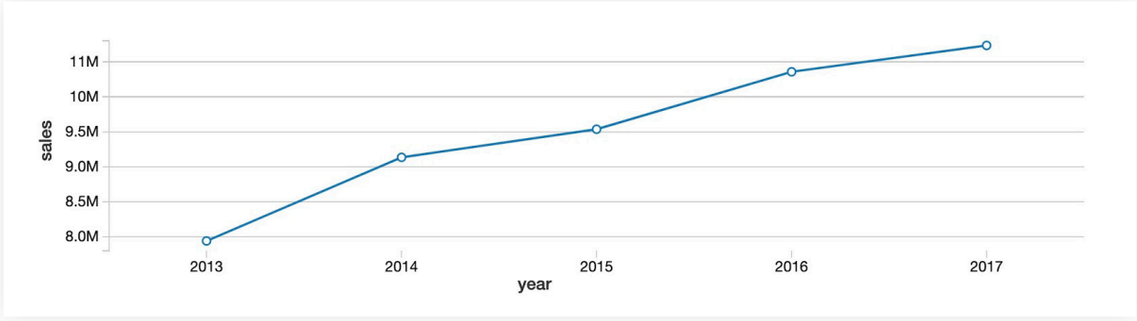
새로운 기능(예: Facebook Prophet에서 제공하는 기능)이 데이터 사이언스 계에 나타나고 있고, 기업들은 이런 머신 러닝 모델을 시계열 예측 요구 사항에 적용할 유연성을 얻고자 합니다.

리테일러 등이 기존 예측 솔루션에서 벗어나려면 조직 내부에서 수요 예측의 복잡성에 대한 전문성을 기르는 동시에, 시기적절하게 수십만, 심지어 수백만 개의 머신 러닝 모델을 생성하는 데 필요한 업무도 효율적으로 분담해야 합니다. 다행히 Spark를 사용하여 이 정도의 모델을 훈련할 수 있으므로 제품과 서비스에 대한 전반적 수요뿐만 아니라 각 위치에 있는 각 제품의 고유 수요도 예측할 수 있습니다.

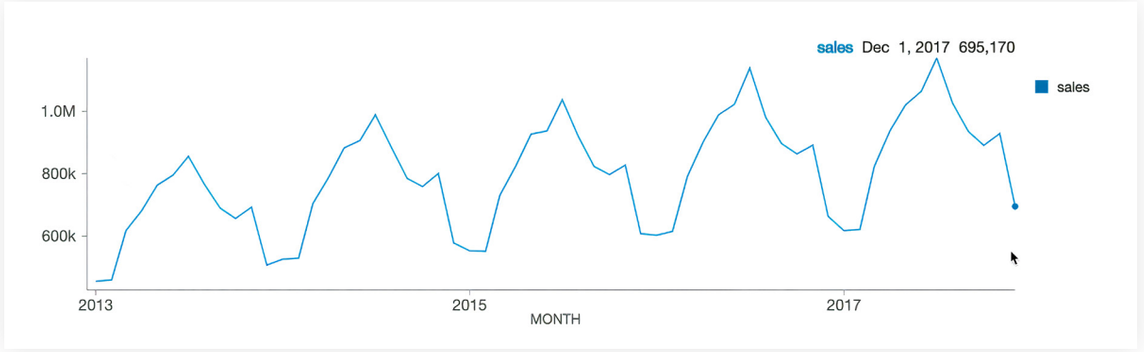
## 시계열 데이터에서 수요 계절성 시각화

Prophet을 사용하여 각 매장과 제품에 세분화된 수요 예측을 생성하는 방법을 보여드리기 위해 Kaggle에서 공개적으로 제공되는 **데이터 세트**를 사용할 것입니다. 이는 10개 매장에 있는 50개 품목에 대한 일일 매출 데이터 5년분으로 구성됩니다.

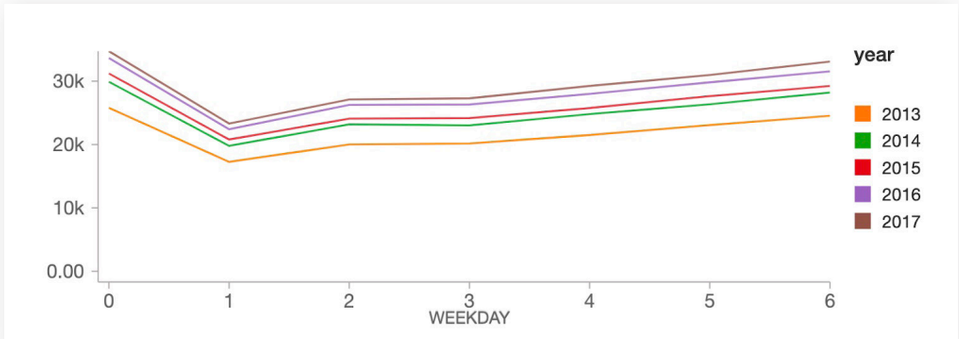
먼저 모든 제품과 매장의 전체 연간 매출 동향을 살펴보겠습니다. 여기에서 볼 수 있듯이, 전체 제품 매출은 매년 늘어나고 있고 일정 수준에서 수렴할 기미는 보이지 않습니다.




이제 같은 데이터를 월별로 살펴보면 매년 상승하는 흐름이 월 단위로는 일정하게 나타나지 않는 것을 알 수 있습니다. 그 대신 여름에는 계절적인 성수기 패턴이, 겨울에는 비수기 패턴이 명확하게 드러납니다. **Databricks Collaborative Notebooks**에 구축된 데이터 시각화 기능을 사용하면 그래프에 마우스를 가져가서 각 월의 데이터 값을 확인할 수 있습니다.




요일별로는 일요일(0일 차)에 매출이 가장 높았다가, 월요일(1일 차)에 급격히 하락한 다음, 주말이 올 때까지 서서히 회복됩니다.



## Facebook Prophet에서 간단한 시계열 예측 모델을 사용해보겠습니다.

위의 그래프에서 볼 수 있듯이, 데이터에서는 매년 꾸준하게 매출이 상승하고 있고 연간, 주간 계절적 패턴을 동반합니다. Prophet은 이런 데이터의 중첩 패턴을 찾도록 설계되었습니다.

Facebook Prophet은 scikit-learn API를 따르기 때문에 sklearn을 사용해본 경험이 있는 사람이라면 누구나 쉽게 익힐 수 있습니다. 2열 pandas DataFrame을 입력값으로 전달해야 합니다. 첫 번째 열은 날짜이고 두 번째 열은 예측하기 위한 값(이 경우, 매출)입니다. 데이터 형식이 적절할 경우 쉽게 모델을 구축할 수 있습니다.

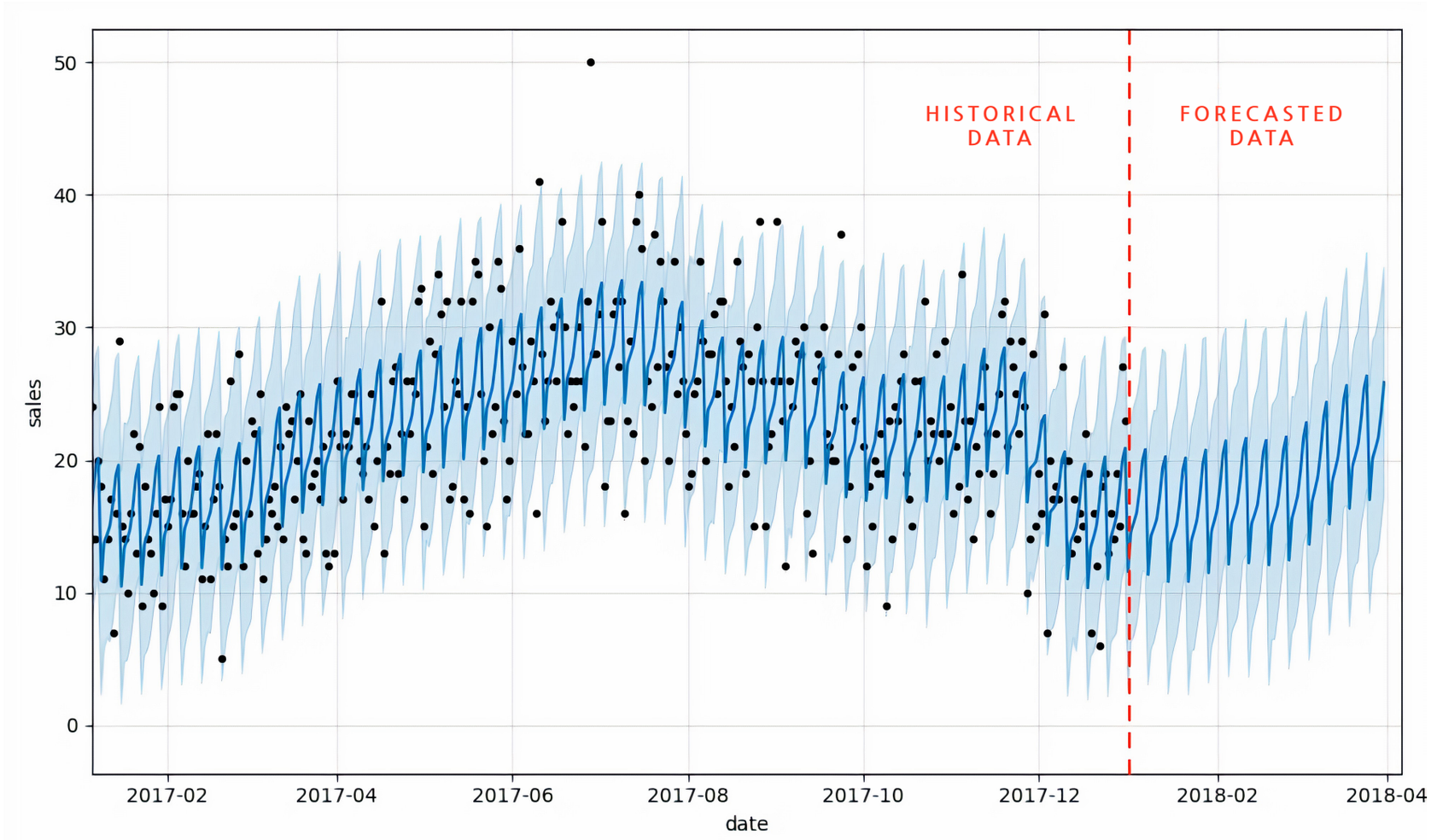
```
import pandas as pd
from fbprophet import Prophet

# instantiate the model and set parameters
model = Prophet(
    interval_width=0.95,
    growth='linear',
    daily_seasonality=False,
    weekly_seasonality=True,
    yearly_seasonality=True,
    seasonality_mode='multiplicative'
)

# fit the model to historical data
model.fit(history_pd)
```

이제 모델을 데이터에 피팅했으므로 90일 예측을 구축해보겠습니다. 아래의 코드에서 prophet의 `make_future_dataframe` 메서드를 사용하여 과거 날짜와 90일 이전을 포함하는 데이터 세트를 정의합니다.

```
future_pd = model.make_future_dataframe(
    periods=90,
    freq='d',
    include_history=True
)

# predict over the dataset
forecast_pd = model.predict(future_pd)
```

이것으로 끝입니다! Prophet의 내장 .plot 메서드를 사용하여 실제 및 예측 데이터가 어떻게 나타나고, 미래가 어떻게 예측되는지 시각화할 수 있습니다. 보다시피, 앞서 설명했던 주간 및 계절적 수요 패턴이 예측 결과에 반영되어 있습니다.

```
predict_fig = model.plot(forecast_pd, xlabel='date', ylabel='sales')
display(fig)
```




이 시각화 자료는 다소 복잡해 보입니다. Bartosz Mikulski가 **훌륭한 분석**을 제공하므로 한 번 확인해보시는 것이 좋습니다.
요약해서 말씀드리면, 검은색 점은 실제 데이터이고 남색 선은 예측을 나타냅니다. 하늘색 띠는 (95%) 불확도를 나타냅니다.

## Prophet과 Spark를 사용하여 수백 개의 시계열 예측 모델 훈련

하나의 시계열 예측 모델을 구축하는 방법을 알아보았으므로, Apache Spark를 사용하여 모델을 늘려보겠습니다. 이번 목표는 전체 데이터 세트에 하나의 예측을 생성하는 것이 아니라, 각 제품-매장 조합에 대해 수백 개의 모델과 예측을 생성하는 것입니다. 하나씩 작업하려면 엄청난 시간이 소요될 것입니다.

이런 방식으로 모델을 구축하면 (가령) 슈퍼마켓 체인은 선더스키 매장과 클리블랜드 매장의 서로 다른 우유 주문량을 각 위치의 수요에 따라 정밀하게 예측할 수 있습니다.

### Spark DataFrames을 사용하여 시계열 데이터 처리 배포

데이터 사이언티스트는 Apache Spark와 같은 분산된 데이터 처리 엔진을 사용하여 대량의 모델을 훈련하는 문제를 해결해야 하는 경우가 많습니다. Spark 클러스터를 사용하면 해당 클러스터에 있는 각 작업자 노드는 다른 작업자 노드와 동시에 모델의 하위 집합을 훈련할 수 있어서 전체 시계열 모델을 훈련하는 데 필요한 시간을 대폭 절약할 수 있습니다.

물론, 작업자 노드(컴퓨터)로 구성된 클러스터 하나에서 모델을 훈련하려면 더 많은 클라우드 인프라가 필요하고 여기에는 비용이 발생합니다. 하지만 온디맨드 클라우드 리소스를 손쉽게 사용할 수 있으므로 기업에서는 필요한 리소스를 재빨리 프로비저닝하여 모델을 훈련하고, 똑같이 빠른 속도로 리소스를 해제할 수도 있습니다. 따라서 물리적 자산에 장기적으로 투자하지 않더라도 엄청난 확장성을 달성할 수 있습니다.

Spark에서 분산된 데이터 처리를 지원하는 핵심 메커니즘은 DataFrame입니다. Spark DataFrame으로 데이터를 로드하면 클러스터의 작업자 전체에 데이터가 분산됩니다. 그러면 작업자가 동시에 데이터 하위 집합을 처리함으로써, 작업을 수행하는 데 필요한 전반적인 시간이 감소합니다.

물론, 각 작업자는 처리에 필요한 데이터 하위 집합에 액세스해야 합니다. 키 값(이 경우, 매장과 품목의 조합)에 대해 데이터를 그룹화하여 해당 키 값에 대한 모든 시계열 데이터를 특정 작업자 노드로 통합합니다.

```
store_item_history
    .groupBy('store', 'item')
    # . . .
```

> 얼마나 효율적으로 대량의 모델을 훈련할 수 있는지 보여드리기 위해 groupBy 코드를 공유합니다. 그러나 다음 섹션에서 UDF를 설정하고 데이터에 적용해야 실제로 어떨지 알 수 있습니다.

## pandas 사용자 정의 함수 활용

매장과 품목을 기준으로 시계열 데이터를 적절히 그룹화하면 각 그룹에 하나의 모델을 훈련해야 합니다. 이를 위해서는 pandas 사용자 정의 함수(UDF)를 사용합니다. DataFrame에서 사용자 정의 함수를 각 데이터 그룹에 적용할 수 있습니다.

이 UDF는 각 그룹에 대해 모델을 훈련할 뿐만 아니라, 해당 모델에서 예측을 나타내는 결과 세트도 생성합니다. 이 함수는 다른 함수와 독립적으로 DataFrame에서 각 그룹에 대해 훈련하고 예측을 생성하지만, 각 그룹에서 반환된 결과를 하나의 DataFrame 결과로 편리하게 통합할 수 있습니다. 그러면 매장-품목 단위 예측을 생성하면서도 하나의 결과 데이터 세트로 분석 전문가와 관리자에게 제시할 수 있습니다.

축약된 Python 코드에서 볼 수 있듯이, UDF를 구축하는 방법은 비교적 간단합니다. UDF는 `pandas_udf` 메서드로 인스턴스화되고, 반환할 데이터 스키마와 수신할 데이터 타입을 식별합니다. 그다음에는 UDF의 작업을 실행할 함수를 정의하겠습니다.

함수를 정의하는 동안 모델을 인스턴스화해서 구성하고, 수신한 데이터에 피팅할 것입니다. 모델에서 예측하면 그 데이터는 함수의 결과값으로 반환됩니다.

```
@pandas_udf(result_schema, PandasUDFType.GROUPED_MAP)
def forecast_store_item(history_pd):

    # instantiate the model, configure the parameters
    model = Prophet(
        interval_width=0.95,
        growth='linear',
        daily_seasonality=False,
        weekly_seasonality=True,
        yearly_seasonality=True,
        seasonality_mode='multiplicative'
    )

    # fit the model
    model.fit(history_pd)

    # configure predictions
    future_pd = model.make_future_dataframe(
        periods=90,
        freq='d',
        include_history=True
    )

    # make predictions
    results_pd = model.predict(future_pd)

    # . . .

    # return predictions
    return results_pd
```

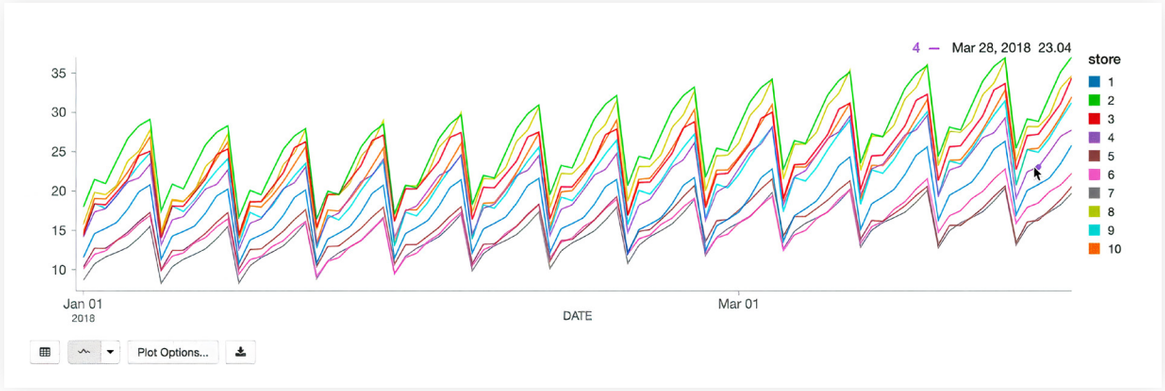
이제 모든 결과를 합치기 위해 앞서 설명한 `groupBy` 명령을 사용하여 데이터 세트가 각 매장과 품목 조합을 대표하는 그룹으로 분할되었는지 확인합니다. 이제 UDF를 DataFrame에 `적용` 하면 UDF에서 모델을 찾고 각 데이터 그룹에 대해 예측할 수 있습니다.

함수를 각 그룹에 적용하여 반환된 데이터 세트는 예측을 생성한 날짜를 반영하여 업데이트됩니다. 최종적으로 기능을 프로덕션으로 배포했을 때 각 모델 런에서 생성된 데이터를 추적하는 데 도움이 됩니다.

```
from pyspark.sql.functions import current_date

results = (
store_item_history
.groupBy('store', 'item')
.apply(forecast_store_item)
.withColumn('training_date', current_date())
)
```

## 다음 단계

이제 각 매장-품목 조합에 대해 시계열 예측 모델을 생성했습니다. 분석 전문가는 SQL 쿼리를 사용하여 각 제품에 대한 맞춤형 예측을 확인할 수 있습니다. 아래의 그래프는 10개 매장에서 제품 #1에 대한 예상 수요를 나타냅니다. 보다시피, 수요 예측은 매장마다 다르지만 일반적 패턴은 모든 매장에서 일치하는 것은 예상한 그대로입니다.




새로운 매출 데이터가 도착하면 새로운 예측을 효율적으로 생성하고 기존 표 구조에 첨부할 수 있습니다. 분석 전문가는 상황에 맞춰 기업의 기대를 업데이트할 수 있습니다.

자세한 내용은 **Starbucks에서 Facebook Prophet과 Azure Databricks를 사용하여 대규모 수요를 예측하는 방법** 온디맨드 웨비나를 참조하세요.

7장:

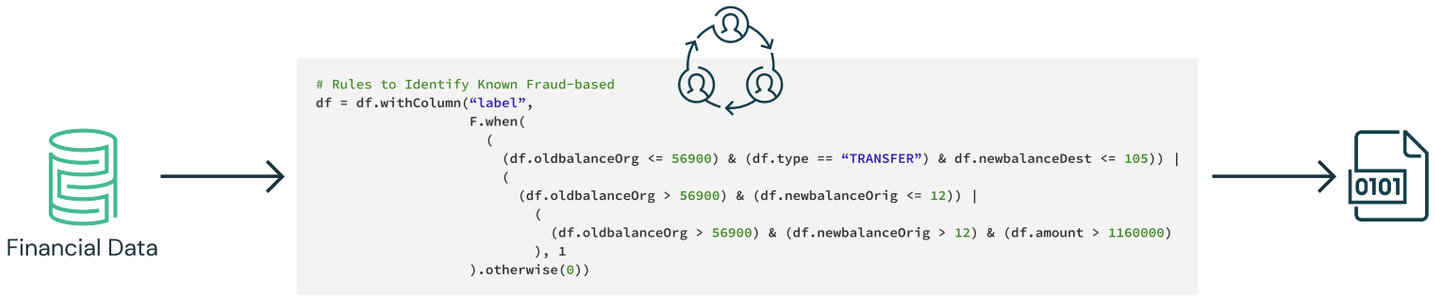
# Databricks에서 결정 트리 및 MLflow로 대규모 금융 사기 탐지

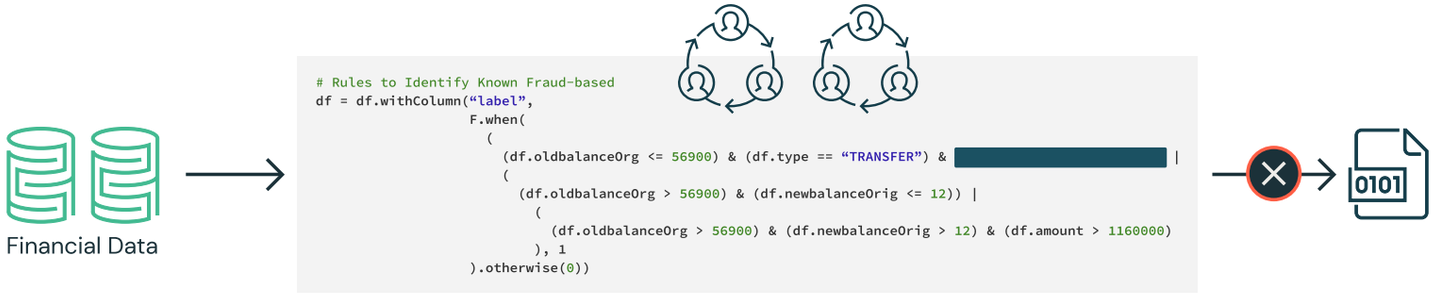
글: Elena Boiarskaia, Navin Albert 및 Denny Lee

2019년 5월 02일

Databricks에서 이 노트북을 확인해보세요 →

어떤 사용 사례에서나 인공 지능을 사용하여 대규모 사기 패턴을 탐지하기란 어렵습니다. 대량의 과거 데이터를 검토해야 하고, 머신 러닝과 딥러닝 기술이 끊임없이 발전하면서 복잡성이 커지는 데다 사기 행위에 대한 매우 적은 실제 사례만으로 바늘이 어떻게 생겼는지도 모르면서 건초에서 바늘 찾듯이 찾아야 합니다. 금융 서비스 산업에서 보안에 대한 우려와 더불어 사기 행위를 찾아내는 방법을 설명해야 할 중요성이 커지면서 이 작업은 더욱 복잡해졌습니다.

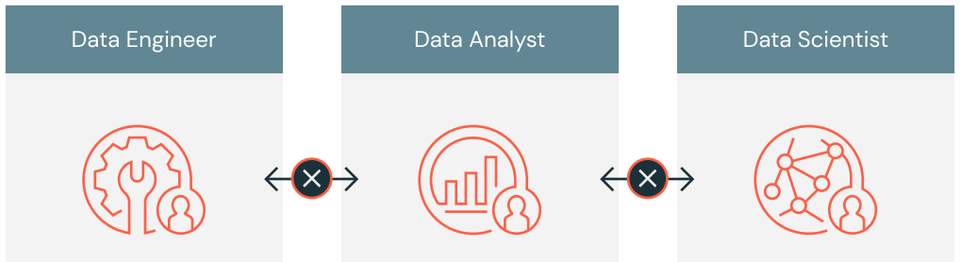


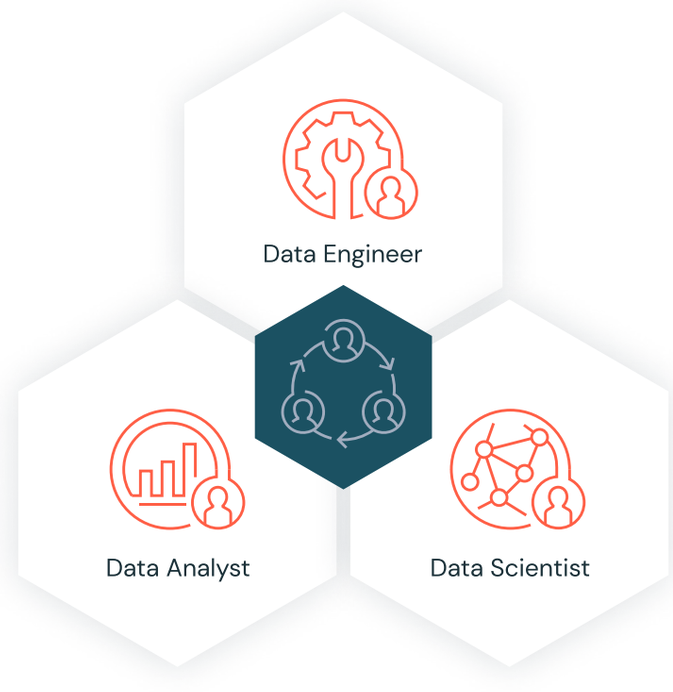

이런 탐지 패턴을 구축하기 위해 해당 분야 전문가팀이 사기범의 일반적인 행동을 바탕으로 규칙 세트를 만들었습니다. 워크플로에는 금융 사기 탐지 분야의 주제 전문가가 참여하여 특정 행동에 대한 요구 사항들을 작성할 수도 있습니다. 데이터 사이언티스트는 사용할 수 있는 데이터의 하위 샘플을 받아서 이러한 요구 사항과 일부 알려진 사기 사건을 활용하여 딥러닝 또는 머신 러닝 알고리즘 세트를 선택합니다. 데이터 엔지니어는 그 결과로 얻은 모델을 임계값을 적용한 규칙 세트로 변환하여 프로덕션에 패턴을 배포합니다. 이는 대부분 SQL을 사용하여 구현합니다.

databricks

이 방법을 사용하는 금융 기관은 명확한 특징들을 파악하고, 이를 활용해 일반 데이터 보호 규정(GDPR)에 따라 사기성 거래를 찾아낼 수 있습니다. 그러나 여러 가지 어려움도 수반됩니다. 하드코딩된 규칙 세트로 사기 탐지 시스템을 구현하면 매우 불안정합니다. 사기 패턴이 바뀌었을 때 업데이트에 매우 오랜 시간이 걸리게 됩니다. 그러면 현재 시장에서 발생하는 사기성 행위의 변화를 따라잡고, 이에 적응하기 어렵게 됩니다.



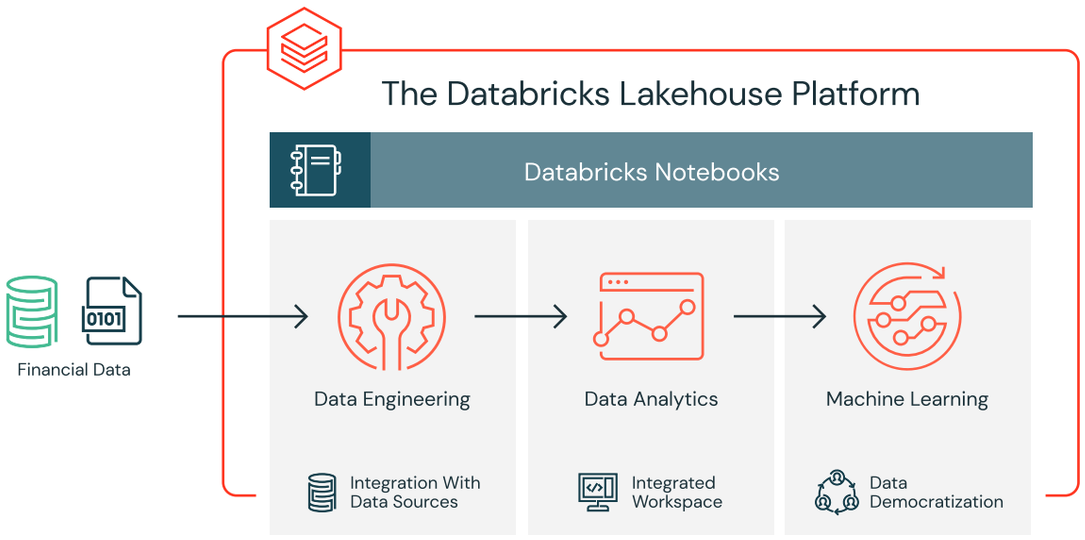

게다가 앞서 설명한 워크플로의 시스템은 대개 사일로화되어 있고 분야 전문가, 데이터 사이언티스트, 데이터 엔지니어가 모두 제각각 일합니다. 데이터 엔지니어는 엄청난 양의 데이터를 관리하고 분야 전문가와 데이터 사이언티스트의 작업을 프로덕션 코드로 변환해야 합니다. 공통적인 플랫폼이 없기 때문에 분야 전문가와 데이터 사이언티스트는 컴퓨터 하나에 들어갈 정도로 추출된 데이터를 사용해서 분석해야 합니다. 따라서 소통에 난맥이 생기고, 궁극적으로는 협업이 어려워집니다.




이 블로그에서는 Databricks 플랫폼에서 이런 여러 가지 규칙 기반 탐지 사용 사례를 머신 러닝 사용 사례로 변환하고, 사기 탐지 분야의 주요 담당자(분야 전문가, 데이터 사이언티스트, 데이터 엔지니어)를 통합하는 방법을 살펴봅니다. 머신 러닝 사기 탐지 데이터 파이프라인을 만들고, 프레임워크를 실시간 활용하여 데이터를 시각화함으로써 대규모 데이터 세트에서 모듈식 기능을 구축하는 방법을 배울 것입니다. 또한, 결정 트리와 Apache Spark™ MLlib를 사용한 사기 탐지 방법을 알아보겠습니다. 그런 다음, MLflow로 모델을 반복 개선하고 정확도를 높일 것입니다.

## 머신 러닝으로 문제 해결

금융 산업에서는 머신 러닝 모델과 관련하여 다소 꺼려하는 분위기가 있습니다. 사기 사례를 발견하더라도 근거를 제공하지 못하는 "블랙박스"라고 생각하기 때문입니다. 데이터 사이언스를 활용하는 것은 GDPR 요구 사항과 금융 규제로 인해 불가능에 가까운 영역으로 보입니다. 그러나 여러 건의 성공적인 사용 사례에서 볼 수 있듯이, 머신 러닝을 대규모 사기 탐지에 적용하면 앞서 설명한 여러 가지 문제를 해결할 수 있습니다.




금융 사기를 탐지하기 위한 지도 머신 러닝 모델을 훈련하는 것은 실제 사기 행각이 발각된 사례가 적어서 매우 어렵습니다. 그러나 특정 사기 타입을 식별하는 규칙 세트가 알려져 있기 때문에 합성 레이블 세트와 최초 feature set을 만드는 데는 도움이 됩니다. 이 분야의 전문가가 개발한 탐지 패턴의 결과값은 적절한 승인 절차를 거쳐 프로덕션에 배포되었을 것입니다. 예상되는 사기 행위 플래그를 생성하므로 이를 시작점으로 삼아 머신 러닝 모델을 훈련할 수 있습니다.

이 방법은 세 가지 우려 사항을 동시에 완화합니다.

1. 바로 훈련 레이블이 부족하고
2. 사용할 feature을 결정해야 하고
3. 모델에 대한 적절한 벤치마크가 있어야 한다는 것입니다.

규칙 기반 사기 행위 플래그를 인식하도록 머신 러닝 모델을 훈련하면 오차 행렬을 통해 예상되는 결과값과 직접 비교할 수 있습니다. 규칙이 규칙 기반 탐지 패턴과 매우 근사하게 일치하는 경우, 회의적인 입장을 취하는 사람들이 머신 러닝 기반 사기 예방의 효과를 신뢰하는 데 도움이 됩니다. 이 모델의 결과값은 매우 해석하기 쉽고, 원래 탐지 패턴과 비교하면 예상 거짓 양성과 거짓 음성에 대한 기본적인 논의가 가능할 수 있습니다.

게다가 머신 러닝 모델을 해석하기 어렵다는 우려는 결정 트리 모델을 최초 머신 러닝 모델로 사용하면 더욱 완화할 수 있습니다. 이 모델은 규칙 세트에 대해 훈련되므로 결정 트리가 다른 어떤 머신 러닝 모델보다도 더 나은 성과를 보일 것입니다. 물론, 모델이 가장 투명하다는 장점도 있습니다. 기본적으로 인간이 개입하거나 규칙이나 임계값을 하드코딩하지 않아도 사기에 대한 의사결정 과정을 보여줄 수 있습니다. 나중에 모델을 반복 개선하는 동안 다른 알고리즘까지 활용하면 정확도를 최대로 높일 수도 있습니다. 궁극적으로 모델의 투명성은 알고리즘에 입력된 feature를 이해해야 가능합니다. 해석할 수 있는 feature가 있으면 해석 가능하고 논리적인 모델 결과를 얻게 됩니다.

머신 러닝 기술의 가장 큰 장점은 처음에 모델링에 노력을 기울이면 나중에 모듈식 반복 개선을 통해 레이블 세트, feature, 모델 타입을 매우 쉽고 매끄럽게 업데이트하고 프로덕션 시간을 단축할 수 있다는 것입니다. 또한, Databricks Collaborative Notebooks에서 이를 지원하면 분야 전문가, 데이터 사이언티스트, 데이터 엔지니어가 대규모로 동일한 데이터 세트를 분석하고 노트북 환경에서 바로 협업할 수 있습니다. 그럼 시작하겠습니다.

## 데이터 입력 및 탐색

이 예시에서는 합성 데이터 세트를 사용할 것입니다. 데이터 세트를 직접 로드하려면 Kaggle에서 로컬 컴퓨터로 다운로드한 다음, Azure 및 AWS에서 데이터 가져오기를 통해 데이터를 가져옵니다.

PaySim 데이터는 구현된 휴대전화 송금 서비스의 거래 로그 1개월분에서 추출한 실제 거래 샘플을 기반으로 휴대전화 송금 거래를 시뮬레이션합니다. 아래의 표는 이 데이터 세트에서 제공하는 정보를 나타냅니다.

| Column Name | Description |
|---|---|
| step | maps a unit of time in the real world. In this case 1 step is 1 hour of time. Total steps 744 (30 days simulation). |
| type | CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER. |
| amount | amount of the transaction in local currency. |
| nameOrig | customer who started the transaction |
| oldbalanceOrg | initial balance before the transaction |
| newbalanceOrig | new balance after the transaction |
| nameDest | customer who is the recipient of the transaction |
| oldbalanceDest | initial balance recipient before the transaction. Note that there is not information for customers that start with M (Merchants). |
| newbalanceDest | new balance recipient after the transaction. Note that there is not information for customers that start with M (Merchants). |

### 데이터 탐색

**Creating the DataFrames: DataFrames 생성**: 데이터가 Databricks File System (DBFS)에 업로드되어 있으므로 Spark SQL을 사용하여 DataFrames를 빠르고 쉽게 생성할 수 있습니다.

```
# Create df DataFrame which contains our simulated financial fraud detection dataset
df = spark.sql("select step, type, amount, nameOrig, oldbalanceOrg, newbalanceOrig,
nameDest, oldbalanceDest, newbalanceDest from sim_fin_fraud_detection")
```

DataFrame을 생성했으므로 스키마와 처음부터 1,000행까지 살펴보면서 데이터를 검토하겠습니다.

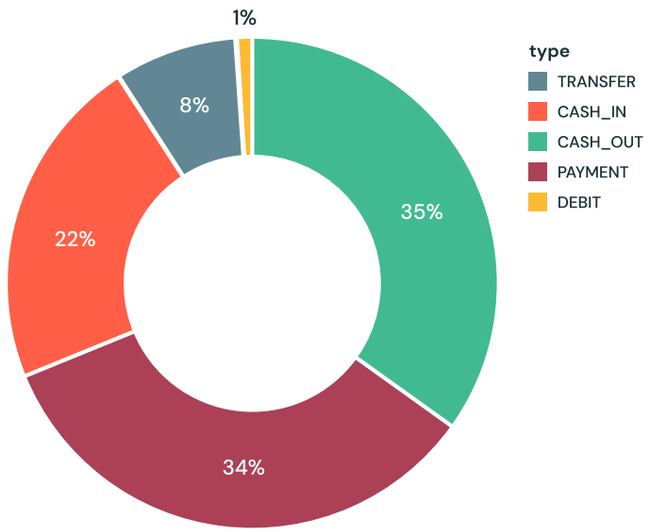
```
# Review the schema of your data
df.printSchema()
root
 |-- step: integer (nullable = true)
 |-- type: string (nullable = true)
 |-- amount: double (nullable = true)
 |-- nameOrig: string (nullable = true)
 |-- oldbalanceOrg: double (nullable = true)
 |-- newbalanceOrig: double (nullable = true)
 |-- nameDest: string (nullable = true)
 |-- oldbalanceDest: double (nullable = true)
 |-- newbalanceDest: double (nullable = true)
```

| step | type | amount | nameOrig | oldbalanceOrg | newbalanceOrig | nameDest | oldbalanceDest |
|---|---|---|---|---|---|---|---|
| 1 | PAYMENT | 9839.64 | C1231006815 | 170136 | 160296.36 | M1979787155 | 0 |
| 1 | PAYMENT | 1864.28 | C1666544295 | 21249 | 19384.72 | M2044282225 | 0 |
| 1 | TRANSFER | 181 | C1305486145 | 181 | 0 | C553264065 | 0 |
| 1 | CASH_OUT | 181 | C840083671 | 181 | 0 | C38997010 | 21182 |
| 1 | PAYMENT | 11668.14 | C2048537720 | 41554 | 29885.86 | M1230701703 | 0 |
| 1 | PAYMENT | 7817.71 | C90045638 | 53860 | 46042.29 | M573487274 | 0 |
| 1 | PAYMENT | 7107.77 | C154988899 | 183195 | 176087.23 | M408069119 | 0 |
| 1 | PAYMENT | 7861.64 | C1912850431 | 176087.23 | 168225.59 | M633326333 | 0 |

## 거래 유형

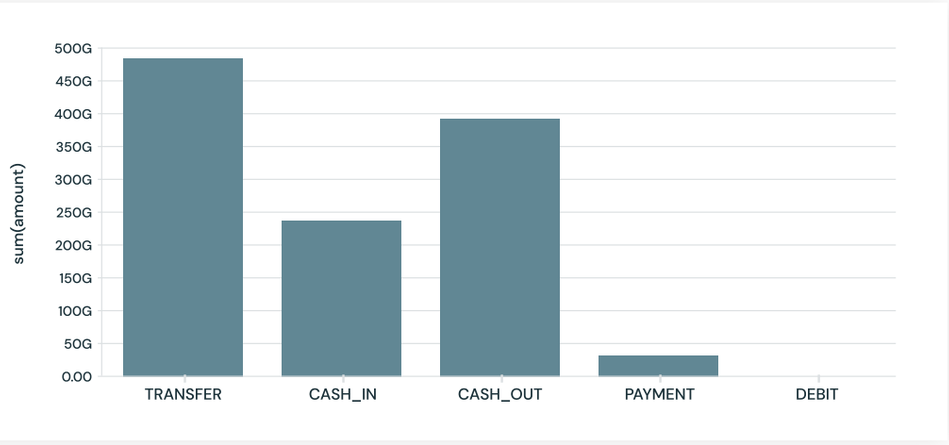
데이터를 시각화하고, 데이터가 캡처하는 거래 유형과 이 거래 유형이 전체 거래량에 미치는 영향에 대해 알아보겠습니다.

```
%sql
-- Organize by Type
select type, count(1) from financials group by type
```




운영 규모가 얼마나 되는지 알아보기 위해 거래 유형과 송금된 현금(즉, sum(amount))에 미치는 영향을 기준으로도 데이터를 시각화해보겠습니다.

```
%sql
select type, sum(amount) from financials group by type
```




## 규칙 기반 모델

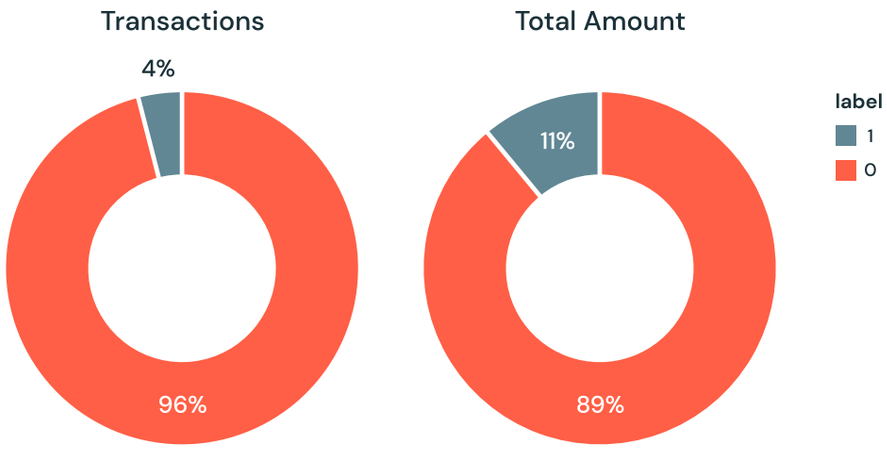
알려진 사기 사례의 데이터가 적어서 이것으로 모델을 훈련하기는 어려울 듯합니다. 이 기술을 실제로 응용하는 대부분의 경우, 분야 전문가가 설정한 규칙 세트로 사기 탐지 패턴을 찾아냅니다. 여기에서는 이 규칙에 따라 'label'이라는 열을 만들겠습니다.

```
# Rules to Identify Known Fraud-based
df = df.withColumn("label",
                   F.when(
                     (
                       (df.oldbalanceOrg <= 56900) & (df.type == "TRANSFER") & (df.
newbalanceDest <= 105)) ¦ ( (df.oldbalanceOrg > 56900) & (df.newbalanceOrig <= 12)) ¦ (
(df.oldbalanceOrg > 56900) & (df.newbalanceOrig > 12) & (df.amount > 1160000)
                             ), 1
                   ).otherwise(0))
```

## 규칙에서 플래그한 데이터 시각화

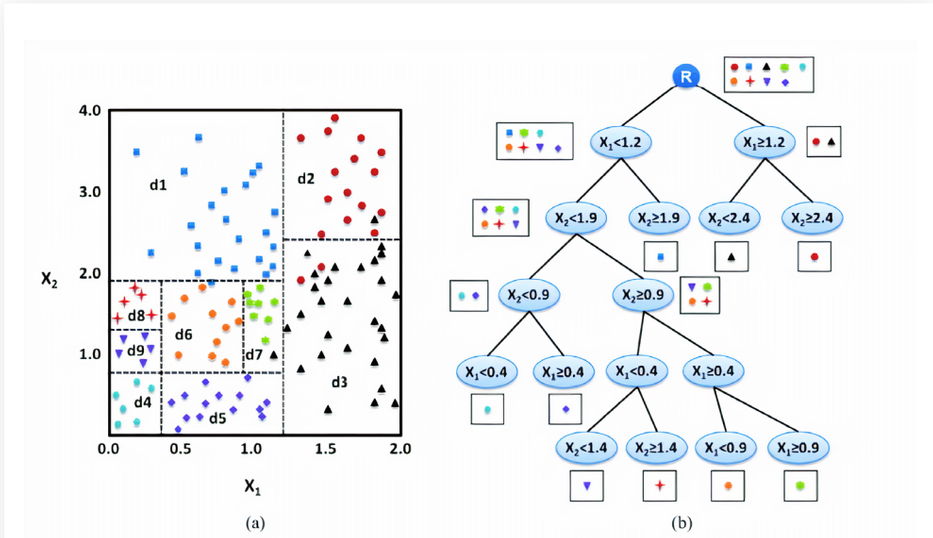
이런 규칙은 상당히 많은 사기 사례를 플래그하는 경우가 많습니다. 플래그된 거래 수를 시각화해보겠습니다. 이 규칙은 사례의 약 4%에 플래그하고, 전체 금액의 11%를 플래그합니다.

```
%sql
select label, count(1) as 'Transactions', sun(amount) as 'Total Amount' from financials_
labeled group by label
```




## 적절한 머신 러닝 모델 선택

대부분 사기 탐지에는 블랙박스 방식을 사용할 수 없습니다. 먼저 분야 전문가는 어떤 거래가 사기로 간주된 이유를 이해해야 합니다. 그런 다음, 법적 조치를 취한다면 증거를 법원에 제출해야 합니다. 결정 트리는 쉽게 해석할 수 있는 모델이고 이 사용 사례의 좋은 시작점입니다. 결정 트리에 대한 자세한 내용은 **"현명한 늙은 나무"** 블로그를 참조하세요.




### 훈련 세트 생성

ML 모델을 구축하고 검증하려면 `.randomSplit`으로 80/20 분할이 필요합니다. 무작위로 선택한 데이터 80%를 훈련에 할당하고 나머지 20%는 결과를 검증하기 위해 남겨둡니다.

```
# Split our dataset between training and test datasets
(train, test) = df.randomSplit([0.8, 0.2], seed=12345)
```

## ML 모델 파이프라인 구축

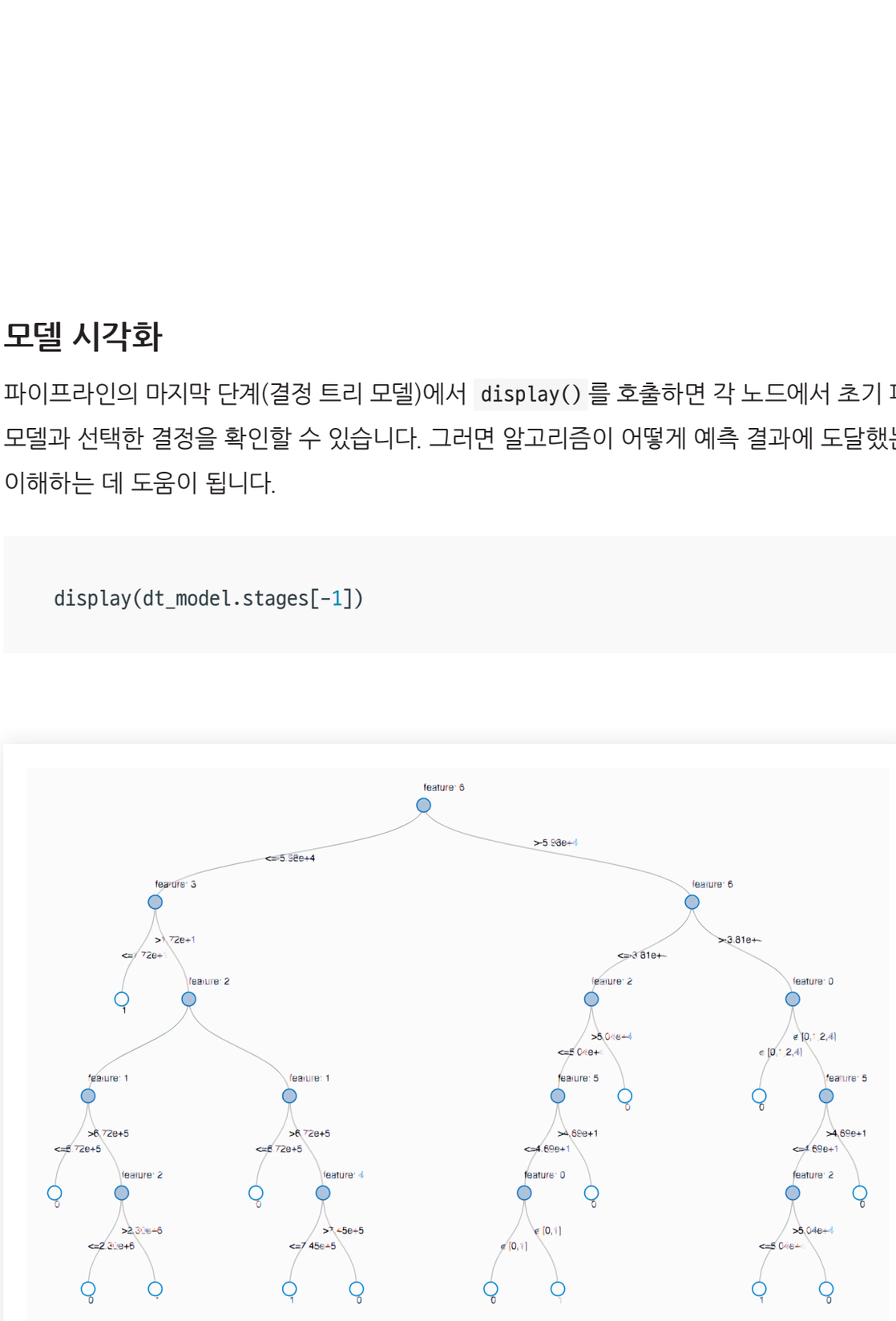
모델에 사용할 데이터를 준비하려면 먼저 `.StringIndexer`를 사용하여 카테고리 변수를 전환해야 합니다. 그런 다음에는 모델에서 사용할 모든 feature들을 모아야 합니다. 결정 트리뿐만 아니라 이런 feature 준비 단계까지 포함하는 파이프라인을 구축해야 다른 데이터 세트에서도 이 단계를 반복할 수 있습니다. 단, 훈련 데이터에 먼저 파이프라인을 피팅하고 나서 테스트 데이터를 변환해야 합니다.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier

# Encodes a string column of labels to a column of label indices
indexer = StringIndexer(inputCol = “type”, outputCol = “typeIndexed”)

# VectorAssembler is a transformer that combines a given list of columns into a single
vector column
va = VectorAssembler(inputCols = [“typeIndexed”, “amount”, “oldbalanceOrg”,
“newbalanceOrig”, “oldbalanceDest”, “newbalanceDest”, “orgDiff”, “destDiff”], outputCol =
“features”)

# Using the DecisionTree classifier model
dt = DecisionTreeClassifier(labelCol = “label”, featuresCol = “features”, seed = 54321,
maxDepth = 5)

# Create our pipeline stages
pipeline = Pipeline(stages=[indexer, va, dt])

# View the Decision Tree model (prior to CrossValidator)
dt_model = pipeline.fit(train)
```

## 모델 시각화

파이프라인의 마지막 단계(결정 트리 모델)에서 `display()`를 호출하면 각 노드에서 초기 피팅 모델과 선택한 결정을 확인할 수 있습니다. 그러면 알고리즘이 어떻게 예측 결과에 도달했는지 이해하는 데 도움이 됩니다.

```
display(dt_model.stages[-1])
```




결정 트리 모델 그림

## 모델 튜닝

최적의 피팅 트리 모델을 얻기 위해서 여러 매개변수 변량으로 모델을 교차 검증할 것입니다. 우리 데이터는 음성 사례 96%와 양성 사례 4%로 구성되어 있으므로, Precision-Recall(PR) 평가 지표를 사용하여 불균형한 분포를 보정하겠습니다.

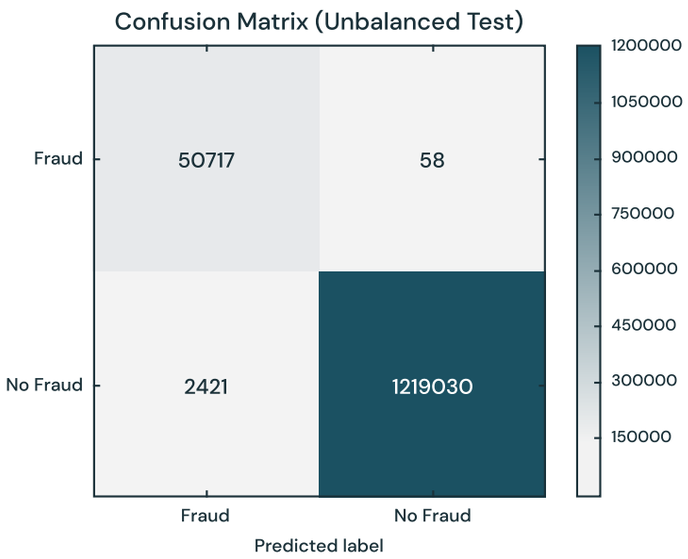
```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Build the grid of different parameters
paramGrid = ParamGridBuilder() \
.addGrid(dt.maxDepth, [5, 10, 15]) \
.addGrid(dt.maxBins, [10, 20, 30]) \
.build()

# Build out the cross validation
crossval = CrossValidator(estimator = dt,
                          estimatorParamMaps = paramGrid,
                          evaluator = evaluatorPR,
                          numFolds = 3)
# Build the CV pipeline
pipelineCV = Pipeline(stages=[indexer, va, crossval])

# Train the model using the pipeline, parameter grid, and preceding
BinaryClassificationEvaluator
cvModel_u = pipelineCV.fit(train)
```

## 모델 성능

우리는 Precision-Recall(PR)과 ROC 곡선(AUC) 지표 아래의 영역을 비교하여 훈련 및 테스트 세트에 대해 모델을 평가합니다. PR과 AUC가 매우 높은 듯이 보입니다.

```
# Build the best model (training and test datasets)
train_pred = cvModel_u.transform(train)
test_pred = cvModel_u.transform(test)

# Evaluate the model on training datasets
pr_train = evaluatorPR.evaluate(train_pred)
auc_train = evaluatorAUC.evaluate(train_pred)

# Evaluate the model on test datasets
pr_test = evaluatorPR.evaluate(test_pred)
auc_test = evaluatorAUC.evaluate(test_pred)

# Print out the PR and AUC values
print(“PR train:”, pr_train)
print(“AUC train:”, auc_train)
print(“PR test:”, pr_test)
print(“AUC test:”, auc_test)

---
# Output:
# PR train: 0.9537894984523128
# AUC train: 0.998647996459481
# PR test: 0.9539170535377599
# AUC test: 0.9984378183482442
```

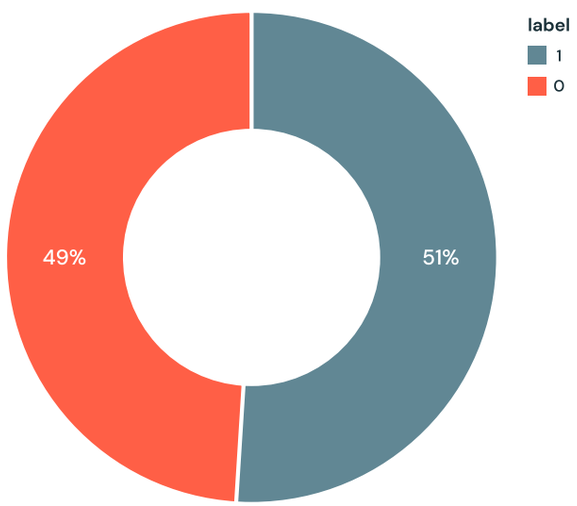
모델이 결과를 어떻게 잘못 분류하는지 알아보기 위해 Matplotlib과 pandas를 사용하여 오차 행렬을 시각화하겠습니다.




## 클래스 균형 조정

이 모델은 원래 규칙에서 찾아낸 것보다 2,421건 이상의 사례를 찾아냅니다. 잠재적 사기 사례를 더 많이 탐지하는 것은 긍정적일 수 있으므로 놀라지 않아도 됩니다. 그러나 원래 규칙에서 탐지되었지만 알고리즘에서 찾아내지 못한 사례가 58건 있습니다. 과소 표집을 사용하여 클래스 균형을 조정하는 방법으로 예측 결과를 개선해보겠습니다. 즉, 모든 사기 사례는 그대로 두고 사기가 아닌 사례만 과소 표집으로 조사하여 그 수치에 매칭시켜 균형 잡힌 데이터 세트를 얻는 것입니다. 새로운 데이터 세트를 시각화했을 때 사기 사례와 그렇지 않은 사례가 50/50이었습니다.

```
## Reset the DataFrames for no fraud (`dfn`) and fraud (`dfy`)
dfn = train.filter(train.label == 0)
dfy = train.filter(train.label == 1)

# Calculate summary metrics
N = train.count()
y = dfy.count()
p = y/N

# Create a more balanced training dataset
train_b = dfn.sample(False, p, seed = 92285).union(dfy)

# Print out metrics
print("Total count: %s, Fraud cases count: %s, Proportion of fraud cases: %s" % (N, y,
p))
print("Balanced training dataset count: %s" % train_b.count())

---
# Output:
# Total count: 5090394, Fraud cases count: 204865, Proportion of fraud cases:
0.040245411258932016
# Balanced training dataset count: 401898
---

# Display our more balanced training dataset
display(train_b.groupBy("label").count())
```



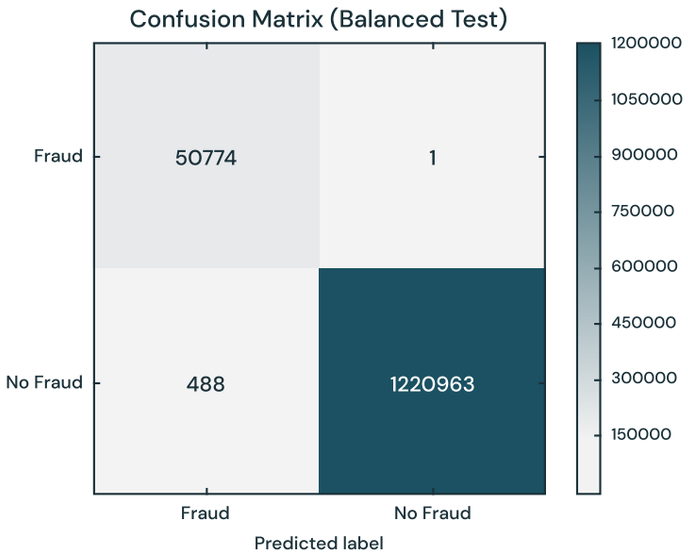
## 파이프라인 업데이트

이제 ML 파이프라인을 업데이트하고 새로운 교차 검증기를 생성해보겠습니다. ML 파이프라인을 사용하기 때문에 새로운 데이터 세트로 업데이트하면 동일한 파이프라인 단계를 빠르게 반복할 수 있습니다.

```
# Re-run the same ML pipeline (including parameters grid)
crossval_b = CrossValidator(estimator = dt,
estimatorParamMaps = paramGrid,
evaluator = evaluatorAUC,
numFolds = 3)
pipelineCV_b = Pipeline(stages=[indexer, va, crossval_b])

# Train the model using the pipeline, parameter grid, and
BinaryClassificationEvaluator using the `train_b` dataset
cvModel_b = pipelineCV_b.fit(train_b)

# Build the best model (balanced training and full test datasets)
train_pred_b = cvModel_b.transform(train_b)
test_pred_b = cvModel_b.transform(test)

# Evaluate the model on the balanced training datasets
pr_train_b = evaluatorPR.evaluate(train_pred_b)
auc_train_b = evaluatorAUC.evaluate(train_pred_b)

# Evaluate the model on full test datasets
pr_test_b = evaluatorPR.evaluate(test_pred_b)
auc_test_b = evaluatorAUC.evaluate(test_pred_b)

# Print out the PR and AUC values
print(“PR train:”, pr_train_b)
print(“AUC train:”, auc_train_b)
print(“PR test:”, pr_test_b)
print(“AUC test:”, auc_test_b)

---
# Output:
# PR train: 0.999629161563572
# AUC train: 0.9998071389056655
# PR test: 0.9904709171789063
# AUC test: 0.9997903902204509
```

## 결과 검토

새로운 오차 행렬의 결과를 살펴보겠습니다. 이 모델은 사기 사례 1개만 잘못 판정했습니다. 클래스 균형을 조정한 덕분에 모델이 개선된 듯합니다.




## 모델 피드백 및 MLflow 사용

프로덕션에 사용할 모델을 선택했다면 모델이 지속해서 찾으려는 행위를 찾아낼 수 있도록 꾸준하게 피드백을 수집해야 합니다. 규칙 기반 레이블로 시작했기 때문에 인간의 피드백을 받아 검증한 실제 레이블을 포함한 추가적인 모델을 제공하고자 합니다. 이 단계는 머신 러닝 프로세스에 대한 신뢰와 확신을 유지하는 데 중요합니다. 분석 전문가가 모든 사례를 검토할 수 없으므로, 모델 결과값을 검증하기 위한 사례를 신중히 선택하여 제공하고자 합니다. 예를 들어 모델의 신뢰도가 낮은 예측은 분석 전문가가 검토하면 좋습니다. 이런 피드백이 있으면 모델을 변화하는 환경에 맞춰 지속해서 개선할 수 있습니다.

MLflow는 이 사이클에서 다양한 모델 버전을 훈련하도록 지원합니다. 실험을 추적하면서 각 모델 구성과 매개변수의 결과를 비교할 수 있습니다. 예를 들어 MLflow UI를 사용하여 균형 잡힌 데이터 세트와 불균형한 데이터 세트에서 훈련한 모델의 PR과 AUC를 비교할 수 있습니다. 데이터 사이언티스트는 MLflow를 사용해서 다양한 모델 지표와 추가적 시각화, 아티팩트를 추적하고 프로덕션에서 어느 모델을 배포할지 결정하는 데 도움을 받을 수 있습니다. 데이터 엔지니어는 선택한 모델과 훈련에 사용할 라이브러리 버전을 프로덕션의 새로운 데이터에 배포할 .jar 파일로 쉽게 가져올 수 있습니다. 따라서 모델 결과를 검토하는 분야 전문가, 모델을 업데이트하는 데이터 사이언티스트, 프로덕션에 모델을 배포하는 데이터 엔지니어는 이런 반복적 과정을 통해 협업이 강화됩니다.

## 결론

Databricks와 MLflow를 사용하여 규칙 기반 사기 탐지 레이블을 머신 러닝 모델로 변환하는 방법의 예시를 살펴보았습니다. 이 방법을 사용하면 확장할 수 있는 모듈식 솔루션을 개발해, 끊임없이 변화하는 사기 행위 패턴을 따라가는 데 도움이 됩니다. 머신 러닝 모델을 구축하여 사기를 찾아내면 모델을 발전시키고 새로운 잠재적 사기 패턴을 찾아내는 피드백 루프를 구축할 수 있습니다. 특히, 결정 트리 모델은 상호운용성과 정확도가 우수해서 사기 탐지 프로그램에 머신 러닝을 도입하기에 좋은 시작점이 될 수 있다는 것을 확인했습니다.

여기에 Databricks 플랫폼을 사용하는 가장 큰 장점은 데이터 사이언티스트, 엔지니어, 비즈니스 사용자가 프로세스 전반에서 매끄럽게 협력할 수 있다는 것입니다. 데이터 준비, 모델 구축, 결과 공유, 모델을 프로덕션에 배포하는 작업을 모두 동일한 플랫폼에서 처리할 수 있으므로 지금까지와는 차원이 다른 협업을 경험할 수 있습니다. 이 방법은 사일로화되어 있던 팀에서 신뢰를 키우고 동적이고 효과적인 사기 탐지 프로그램을 만들 수 있습니다.

몇 분만 투자해서 무료 체험에 등록하여 **이 노트북**을 체험해보고 자신의 모델을 만들어 보세요.

databricks

8장:

# Virgin Hyperloop One에서 Koalas로 처리 시간을 몇 시간에서 몇 분으로 단축한 비결

## pandas 코드를 Apache Spark™으로 매끄럽게 전환하기 위한 실무 가이드

글: **Patryk Oleniuk** 및 **Sandhya Raghavan**

2019년 8월 22일

Virgin Hyperloop One에서는 매우 저렴한 이동 비용으로 승객과 화물을 항공기로 운반할 수 있도록 Hyperloop를 구현해보겠습니다. 우리는 상업적으로 사용할 수 있는 시스템을 구축하기 위해 Devloop Test Track 런, 다양한 테스트 리그, 시뮬레이션, 인프라, 사회경제 데이터를 비롯한 다양하고 방대한 데이터를 수집하고 분석합니다. 이런 데이터를 처리하는 대부분 스크립트는 Python 라이브러리를 사용하여 작성하고, pandas를 기본 데이터 처리 도구로 삼아 모든 것을 결합합니다. 이 블로그 게시물에서는 Koalas를 사용하여 데이터 분석을 확장하고, 약간의 코드 변경만으로 엄청나게 시간을 단축한 사례를 공유하고자 합니다.

새로운 것을 발전시키고 구축하는 만큼, 데이터 처리 요구 사항도 생겨납니다. 데이터 작업의 규모와 복잡성이 커져서 pandas 기반 Python 스크립트는 너무 느려서 비즈니스 요구 사항을 충족할 수 없게 되었습니다. 그래서 처리 시간을 단축하고 유연한 데이터 스토리지와 온디맨드 확장성을 사용하려고 Spark로 눈을 돌리게 되었습니다. 그러나 “Spark로 전환”하는 데는 어려움이 따랐습니다. pandas 기반 코드베이스를 PySpark로 옮기려면 많은 것을 직접적으로 변경해야 했습니다. 우리에게는 훨씬 빠를 뿐만 아니라 되도록 코드를 다시 작성할 필요가 없는 솔루션이 필요했습니다. 그래서 다른 옵션을 조사하였고 이런 번거로운 작업을 건너뛸 수 있는 쉬운 방법이 존재한다는 것을 알고 기뻤습니다. 바로, 얼마 전에 Databricks에서 오픈 소스로 공개한 Koalas 패키지였습니다.

Koalas Readme에 나와 있듯이,

> Koalas 프로젝트는 Apache Spark를 기반으로 pandas DataFrame API를 구현하여 빅데이터와의 상호작용 시 데이터 사이언티스트의 생산성을 향상합니다.
>
> (…)
>
> 이미 pandas에 친숙하다면 새로 배울 필요 없이 Spark를 바로 효과적으로 활용할 수 있습니다.
>
> **하나의 코드베이스를 pandas(테스트, 소규모 데이터 세트)와 Spark(분산된 데이터 세트)에 모두 사용할 수 있습니다.**

이 글에서는 이 말이 (대체로) 사실이라는 것을 입증하고 Koalas를 사용하면 좋은 이유를 설명하려고 합니다. 우리는 pandas 코드를 1% 미만으로 변경하고 Koalas와 Spark에서 코드를 실행할 수 있었습니다. 실행 시간을 10배 이상 단축해 몇 시간에서 몇 분으로 줄였고, 환경을 수평적으로 확장할 수 있으므로 더 많은 데이터를 준비할 수 있었습니다.

## 빠른 시작

Koalas를 설치하기 전에 Spark 클러스터를 설정하고 PySpark에 사용하세요. 그 후 다음 명령을 실행하세요.

```
pip install koalas
```

conda 사용자:

```
conda install koalas -c conda-forge
```

자세한 내용은 Koalas Readme를 참조하세요.

```
import databricks.koalas as ks
kdf = ks.DataFrame({'column1':[4.0, 8.0]}, {'column2':[1.0, 2.0]})
kdf
```

Cmd 1

```
1 import databricks.koalas as ks
2 kdf = ks.DataFrame({'column1':[4.0, 8.0], 'column2':[1.0, 2.0]})
3 kdf
```

▸ (2) Spark Jobs

| | column1 | column2 |
|---|---|---|
| 1 | 8.0 | 2.0 |
| 0 | 4.0 | 1.0 |

Command took 2.13 seconds -- by patryk.oleniuk@hyperloop-one.com at 8/8/2019, 12:40:05 PM on ML Analytics Cluster

보다시피, Koalas는 pandas와 유사한 인터랙티브 테이블을 렌더링할 수 있습니다. 얼마나 편리한지 모릅니다.

## 기본 작업 예시

이 문서에서는 열 4개로 구성된 테스트 데이터를 생성하고 행 개수를 매개변수화하였습니다.

```
import pandas as pd
## generate 1M rows of test data
pdf = generate_pd_test_data( 1e6 )
pdf.head(3)
>>>     timestamp pod_id trip_id speed_mph
0 7.522523 pod_13 trip_6 79.340006
1 22.029855 pod_5 trip_22 65.202122
2 21.473178 pod_20 trip_10 669.901507
```

**책임 부인:** Hyperloop 주제와 관련하여 성능을 평가하기 위해 무작위로 생성한 테스트 파일이며, 우리 데이터를 나타내지 않습니다. 이 문서에서 사용한 전체 테스트 스크립트는 다음의 주소에서 확인할 수 있습니다.

다음과 같이 모든 pod-trip에 대한 몇 가지 중요한 설명적 분석을 평가해보겠습니다. 예를 들어, pod-trip당 이동 시간은 얼마일까요?

필요한 작업:

1. `['pod_id','trip_id']` 로 그룹화
2. 각 이동에 대해 `trip_time` 을 마지막 타임스탬프-첫 번째 타임스탬프로 계산
3. pod-trip 이동 분포 계산(평균, 표준 편차)

**짧고 느린 (pandas) 방법** (코드 조각 #1)

```
import pandas as pd
# take the grouped.max (last timestamp) and join with grouped.min (first timestamp)
gdf = pdf.groupby(['pod_id','trip_id']).agg({'timestamp': ['min','max']})
gdf.columns = ['timestamp_first','timestamp_last']
gdf['trip_time_sec'] = gdf['timestamp_last'] - gdf['timestamp_first']
gdf['trip_time_hours'] = gdf['trip_time_sec'] / 3600.0
# calculate the statistics on trip times
pd_result = gdf.describe()
```

**길고 빠른 (PySpark) 방법** (코드 조각 #2)

```
import pyspark as spark
# import pandas df to spark (this line is not used for profiling)
sdf = spark.createDataFrame(pdf)
# sort by timestamp and groupby
sdf = sdf.sort(desc('timestamp'))
sdf = sdf.groupBy("pod_id", "trip_id").agg(F.max('timestamp').alias('timestamp_last'),
F.min('timestamp').alias('timestamp_first'))
# add another column trip_time_sec as the difference between first and last
sdf = sdf.withColumn('trip_time_sec', sdf2['timestamp_last'] - sdf2['timestamp_first'])
sdf = sdf.withColumn('trip_time_hours', sdf3['trip_time_sec'] / 3600.0)
# calculate the statistics on trip times
sdf4.select(F.col('timestamp_last'),F.col('timestamp_first'),F.col('trip_time_sec'),F.
col('trip_time_hours')).summary().toPandas()
```

**짧고 빠른 (Koalas) 방법** (코드 조각 #3)

```
import databricks.koalas as ks
# import pandas df to koalas (and so also spark)
(this line is not used for profiling)
kdf = ks.from_pandas(pdf)
# the code below is the same as the pandas version
gdf = kdf.groupby(['pod_id','trip_id']).agg({'timestamp': ['min','max']})
gdf.columns = ['timestamp_first','timestamp_last']
gdf['trip_time_sec'] = gdf['timestamp_last'] - gdf['timestamp_first']
gdf['trip_time_hours'] = gdf['trip_time_sec'] / 3600.0
ks_result = gdf.describe().to_pandas()
```

코드 조각 #1~#3은 코드가 동일하므로 "Spark 전환"이 매끄럽습니다. 대부분 pandas 스크립트의 경우, import pandas databricks.koalas를 pd로 변경할 수 있으며, 일부 스크립트는 약간의 수정만으로 파일을 실행할 수 있습니다. 다만 아래와 같은 제한이 적용됩니다.

**결과**

모든 코드 조각은 동일한 pod-trip-time 결과를 반환하는 것으로 확인되었습니다. pandas와 Spark의 설명과 요약 방법은 여기에서 설명했듯이 다소 다르지만 이는 성능에 큰 영향을 미치지 않습니다.

샘플 결과:

Cmd 7

```
1 ks_result[['summary',   'trip_time_hours']]
```

Out[105]:

| | summary | trip_time_hours |
|---|---|---|
| 0 | count | 625 |
| 1 | mean | 0.5761789650162432 |
| 2 | stddev | 0.004673946270277798 |
| 3 | min | 0.5539411756993352 |
| 4 | 25% | 0.5739794243951338 |
| 5 | 50% | 0.5772501165562476 |
| 6 | 75% | 0.5795291941218781 |
| 7 | max | 0.5831203330956781 |

Command took 0.02 seconds -- by patryk.oleniuk@hyperloop-one.com at 8/8/2019, 1:06:14 PM on ML Analytics Cluster

## 고급 예시: UDF 및 복잡한 작업

이제 동일한 DataFrame으로 더욱 복잡한 문제를 해결하고 pandas와 Koalas 구현에 어떤 차이가 있는지 알아보겠습니다.

목표: pod-trip당 평균 속도 분석:

1. `['pod_id','trip id']`로 그룹화
2. 각 pod-trip에서 속도(time) 그래프 아래의 면적을 찾아서 총 이동 거리를 계산합니다(여기에 설명한 방법).
3. 그룹화된 df를 `timestamp` 열로 정렬합니다.
4. 타임스탬프의 차이를 계산합니다.
5. 이 차이에 속도를 곱하면 해당 시차 동안 이동한 거리를 알 수 있습니다.
6. `distance_travelled`의 합계를 계산하면 pod-trip당 총 이동 거리를 알 수 있습니다.
7. `trip time`을 `timestamp.last` - `timestamp.first`로 계산합니다.
   (이전 단락 참조)
8. `average_speed`를 `distance_travelled` / `trip time`으로 계산합니다.
9. pod-trip 이동 분포 계산(평균, 표준 편차)

이 작업은 사용자 정의 apply 함수와 사용자 정의 함수(UDF)로 구현하기로 했습니다.

**pandas 방식** (코드 조각 #4)

```
import pandas as pd
def calc_distance_from_speed( gdf ):
 gdf = gdf.sort_values('timestamp')
 gdf['time_diff'] = gdf['timestamp'].diff()
 return pd.DataFrame({
  'distance_miles':[ (gdf['time_diff']*gdf['speed_mph']).sum()],
  'travel_time_sec': [ gdf['timestamp'].iloc[-1] - gdf['timestamp'].iloc[0] ]
})
results = df.groupby(['pod_id','trip_id']).apply( calculate_distance_from_speed)
results['distance_km'] = results['distance_miles'] * 1.609
results['avg_speed_mph'] = results['distance_miles'] / results['travel_time_sec'] /
60.0
results['avg_speed_kph'] = results['avg_speed_mph'] * 1.609
results.describe()
```

**pandas 방식** (코드 조각 #5)

```
import databricks.koalas as ks
from pyspark.sql.functions import pandas_udf, PandasUDFType
from pyspark.sql.types import *
import pyspark.sql.functions as F
schema = StructType([
 StructField(“pod_id”, StringType()),
 StructField(“trip_id”, StringType()),
 StructField(“distance_miles”, DoubleType()),
 StructField(“travel_time_sec”, DoubleType())
])
@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def calculate_distance_from_speed( gdf ):
 gdf = gdf.sort_values(‘timestamp’)
 print(gdf)
 gdf[‘time_diff’] = gdf[‘timestamp’].diff()
 return pd.DataFrame({
  ‘pod_id’:[gdf[‘pod_id’].iloc[0]],
  ‘trip_id’:[gdf[‘trip_id’].iloc[0]],
  ‘distance_miles’:[ (gdf[‘time_diff’]*gdf[‘speed_mph’]).sum()],
  ‘travel_time_sec’: [ gdf[‘timestamp’].iloc[-1]-gdf[‘timestamp’].iloc[0] ]
})
sdf = spark_df.groupby(“pod_id”,”trip_id”).apply(calculate_distance_from_speed)
sdf = sdf.withColumn(‘distance_km’,F.col(‘distance_miles’) * 1.609)
sdf = sdf.withColumn(‘avg_speed_mph’,F.col(‘distance_miles’)/ F.col(‘travel_time_sec’)
/ 60.0)
sdf = sdf.withColumn(‘avg_speed_kph’,F.col(‘avg_speed_mph’) * 1.609)
sdf = sdf.orderBy(sdf.pod_id,sdf.trip_id)
sdf.summary().toPandas()
# summary calculates almost the same results as describe
```

**Koalas 방식** (코드 조각 #6)

```
import databricks.koalas as ks
def calc_distance_from_speed_ks( gdf ) -> ks.DataFrame[ str, str, float , float]:
 gdf = gdf.sort_values(‘timestamp’)
 gdf[‘meanspeed’] = (gdf[‘timestamp’].diff()*gdf[‘speed_mph’]).sum()
 gdf[‘triptime’] = (gdf[‘timestamp’].iloc[-1] - gdf[‘timestamp’].iloc[0])
 return gdf[[‘pod_id’,’trip_id’,’meanspeed’,’triptime’]].iloc[0:1]

kdf = ks.from_pandas(df)
results = kdf.groupby([‘pod_id’,’trip_id’]).apply( calculate_distance_from_
speed_ks)
# due to current limitations of the package, groupby.apply() returns c0 .. c3
column names
results.columns = [‘pod_id’, ‘trip_id’, ‘distance_miles’, ‘travel_time_sec’]
# spark groupby does not set the groupby cols as index and does not sort them
results = results.set_index([‘pod_id’,’trip_id’]).sort_index()
results[‘distance_km’] = results[‘distance_miles’] * 1.609
results[‘avg_speed_mph’] = results[‘distance_miles’] / results[‘travel_time_
sec’] / 60.0
results[‘avg_speed_kph’] = results[‘avg_speed_mph’] * 1.609
results.describe()
```

Koalas의 apply 구현은 PySpark의 `pandas_udf`를 기반으로 하기 때문에 스키마 정보가 필요합니다. 그래서 함수의 정의에서 타입 힌트도 정의해야 합니다. 이 패키지의 저자는 새로운 사용자 정의 타입 힌트인 `ks.DataFrame`와 `ks.Series`를 사용했습니다. 안타깝게도 apply 메서드의 기존 구현이 상당히 복잡하고 느려서 같은 결과를 얻는 데 약간의 노력이 필요했습니다(열 이름 변경, groupby 키가 반환되지 않음). 그러나 모든 동작은 패키지 문서에 적절한 설명이 나와 있습니다.

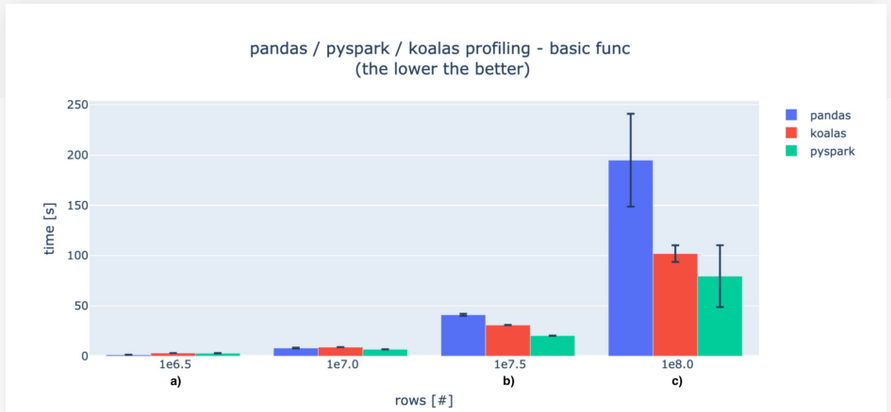
## 성능

Koalas의 성능을 평가하기 위해 각 행에 대한 코드 조각을 프로파일링했습니다.

프로파일링 실험은 다음의 클러스터 구성을 사용하여 Databricks 플랫폼에서 실행했습니다.

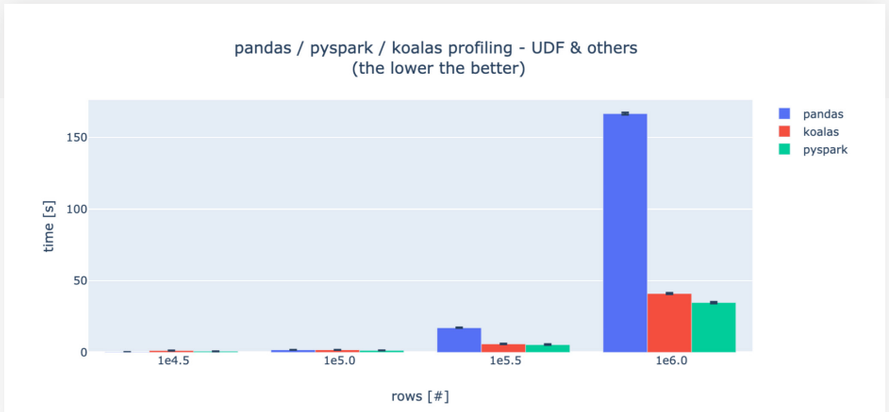
- Spark 드라이버 노드(pandas 스크립트를 실행하는 데도 사용): CPU 코어 8개, 61GB RAM
- 15 Spark 작업자 노드: CPU 코어 4개, 각각 30.5GB RAM(합계: CPU 60개 / 457.5GB)

각 실험은 10번 반복하였고 아래의 클립은 실행의 최소 및 최대 시간을 나타냅니다.




### 기본 작업

데이터 규모가 작을 때는 초기화 작업과 데이터 전송이 연산에 비해 큰 비중을 차지하므로 pandas가 훨씬 빠릅니다(마커 a). 데이터 용량이 큰 경우, pandas의 처리 시간이 분산된 솔루션을 초과합니다(마커 b). Koalas는 다소 성능에 타격이 있지만 데이터가 늘어날수록 PySpark에 가까워집니다(마커 c).




### UDF

UDF 프로파일링은 PySpark와 Koalas 문서에서 지정한 것처럼 성능이 현저하게 저하됩니다. 그렇기 때문에 테스트한 행 수를 기본 작업 사례보다 100배까지 줄여야 합니다. 각 테스트 사례에서 Koalas와 PySpark는 성능 면에서 놀라운 유사성을 보여 구현 면에서 일관성을 나타냅니다. 실험하는 동안 PySpark window 함수를 사용하여 작업 세트를 훨씬 빠르게 실행하는 방법이 존재한다는 것을 확인했지만, Koalas에서는 현재 구현되지 않아 UDF 버전만 비교하기로 했습니다.

## 논의

하나의 노드에서 처리할 수 없는 규모가 큰 데이터 세트에서 pandas 코드를 즉시 확장하고 실행하기를 원한다면 Koalas가 적절합니다. Koalas로 간단히 전환한 후, Spark 클러스터를 확장하기만 하면 더 규모가 큰 데이터 세트를 처리하고 처리 시간을 상당히 개선할 수 있습니다. 성능은 PySpark와 비슷할 것입니다(데이터 세트 규모와 클러스터에 따라 5~50% 낮음).

반면, Koalas API 레이어는 네이티브 Spark에 비해 특히 성능이 눈에 띄게 저하됩니다. 즉, 연산 성능이 중요한 우선순위라면 Python에서 Scala로 변경하는 것이 좋습니다.

## 한계와 차이

Koalas를 사용할 때 처음 몇 시간은 "왜 구현이 되지 않는 거지?!"라는 의문을 느낄 수 있습니다. 현재 이 패키지는 아직 개발 중이고 일부 pandas API 기능이 누락되어 있지만, 대부분 몇 개월 내로 구현될 예정입니다(예: `groupby.diff()` , `kdf.rename()` ).

또한, 프로젝트에 기여한 제 경험에 비추어 보았을 때 일부 기능은 Spark API로 구현하기에는 너무 복잡하거나, 성능 저하가 심해서 건너뛰기도 했습니다. 예를 들어 DataFrame.values는 단일 노드 메모리에 설정된 모든 작업 세트를 구현해야 하는데, 이는 적절하지 않을 뿐만 아니라 불가능한 경우도 있습니다. 대신 드라이브에서 최종 결과를 가져와야 할 때는 `DataFrame.to_pandas()` 또는 `DataFrame.to_numpy()` 를 호출할 수 있습니다.

구현에서 또 한 가지 중요한 점은 Koalas의 실행 체인이 pandas와 다르다는 것입니다. dataframe에서 작업을 실행하면 작업 대기열에 올라가지만 실행되지는 **않습니다**. 결과가 필요할 때(예: `kdf.head()` 또는 `kdf.to_pandas()` )에야 작업이 실행됩니다. pandas는 한 행씩 처리하기 때문에 Spark를 처음 사용하는 사람은 오해할 수 있습니다.

## 결론

Koalas 덕분에 pandas 코드를 "Spark화"하는 부담을 덜었습니다. 여러분도 pandas 코드를 확장하는 데 어려움을 겪고 계신다면 이 방법을 사용해보시기 바랍니다. 어떤 동작이 매우 필요하거나 pandas와 일치하지 않는 점을 발견하면 **문제를 제기**해 주세요. 커뮤니티 차원에서 이 패키지를 적극적이고, 지속해서 개선할 것입니다. 또한, 기여를 해주시는 것도 좋습니다.

## 리소스

1. Koalas GitHub
2. Koalas 문서
3. 이 문서의 코드 조각

> 무료 Databricks 노트북(pandas vs. Koalas)으로 실험을 시작해보세요...

9장:

# Databricks에서 Apache Spark™를 사용하여 개인화된 쇼핑 환경 제공

글: Brett Bevers

2017년 3월 31일

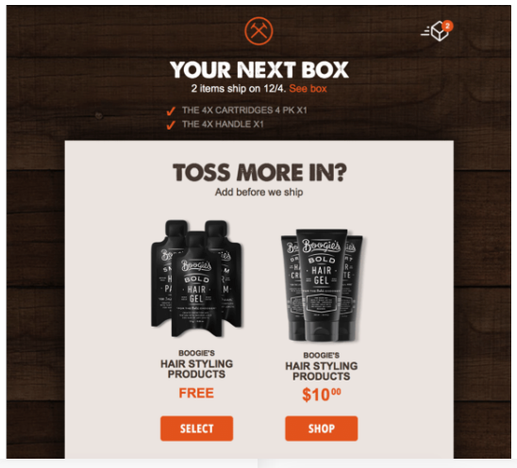
Dollar Shave Club(DSC)은 남성 라이프스타일 브랜드이자 전자상거래 기업으로서, 남성이 면도와 자기 관리 용품을 소비하는 방식에 변화를 일으키고자 합니다. 첨단 사용자 환경을 구현하려면 아마도 데이터가 가장 중요한 자산이 될 것입니다. Databricks는 데이터를 통해 개인화된 고객 환경을 구축하는 데 중요한 파트너가 되어주셨습니다. 이 게시물에서는 Databricks 플랫폼이 어떻게 강력한 사용자 정의 머신 러닝 파이프라인 개발과 배포를 지원하는지 설명합니다.

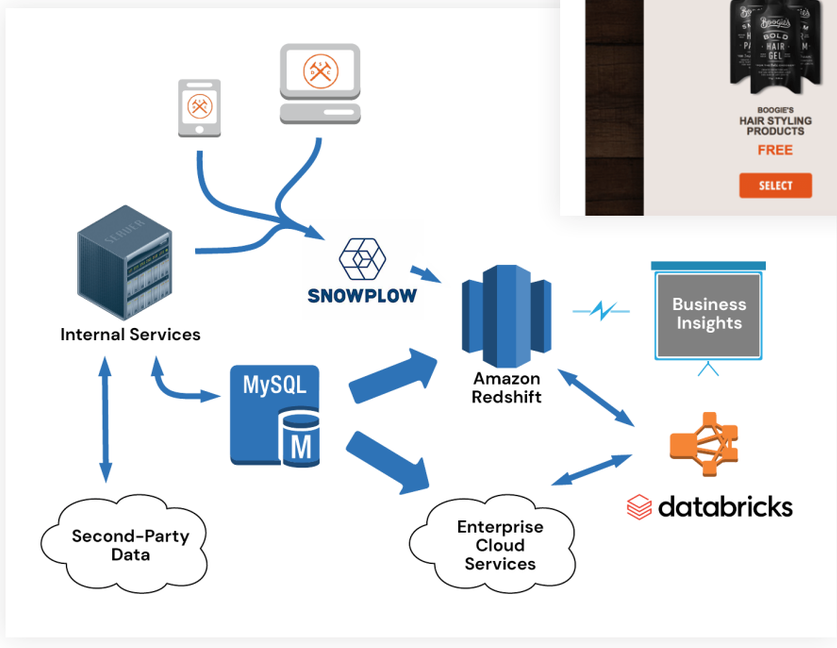
DSC의 기본 서비스는 면도기 카트리지의 월간 구독이고, 상품은 회원에게 바로 배송됩니다. 회원들은 단일 페이지 웹 앱 또는 네이티브 모바일 앱을 통해 계정에 가입하고 이를 관리합니다. 회원은 웹 앱이나 앱을 방문하는 동안 다양한 개인 관리 용품과 세면 용품을 구매할 수 있으며, 브랜드별로 수십 개의 제품이 정리되어 있습니다. 클럽 회원과 게스트는 특징적인 스타일을 좋아하는 사람들을 위해 제작된 오리지널 콘텐츠, 글, 동영상을 즐길 수 있습니다. 청소년기 지루하고 힘든 체육 시간이 떠오르지 않는 문서들로 건강과 자기 관리 주제에 대한 호기심을 충족할 수 있습니다. 스타일, 일, 관계에 대한 간단한 팁을 얻거나, “지구상에서 문명은 얼마나 유지될 수 있는가?”와 같은 중요한 문제에 대한 DSC의 재미있는 해석을 읽을 수도 있습니다. 또한, DSC는 소셜 미디어 채널에서도 소통하고자 하고, 우리 회원들도 열정적으로 참여합니다. DSC는 각 회원이 가장 흥미를 보일 만한 콘텐츠와 상품을 찾아서 더욱 개인화된 우수한 회원 환경을 제공합니다.

## Dollar Shave Club의 데이터

DSC와 회원, 게스트의 상호작용에서 엄청난 양의 데이터가 발생합니다. 우리 엔지니어링팀은 데이터가 회원 환경을 개선할 자산이 될 것을 알고 미리 최신 데이터 인프라에 투자했습니다. 우리 웹 앱에서 내부 서비스와 데이터 인프라는 AWS에서 100% 호스팅됩니다. Redshift 클러스터가 중앙 데이터 웨어하우스 역할을 하며 여러 시스템에서 데이터를 받습니다. 프로덕션 데이터베이스의 기록은 지속해서 웨어하우스에 복제됩니다. 또한, 데이터는 애플리케이션과 오픈 소스 스트리밍 플랫폼인 Apache Kafka에서 조정하는 Redshift 사이를 오갑니다. 우리는 Snowplow(고도의 사용자 정의 오픈 소스 이벤트 파이프라인)를 사용해서 웹과 모바일 클라이언트, 서버측 애플리케이션으로부터 이벤트 데이터를 수집합니다. 클라이언트는 페이지 조회수, 링크 클릭, 탐색 활동, 다양한 사용자 정의 이벤트 및 컨텍스트에 대한 자세한 기록을 전송합니다. 데이터가 Redshift에 도착하면 모니터링, 시각화, 인사이트를 위한 다양한 분석 플랫폼을 통해 액세스할 수 있습니다.

이 정도 수준의 가시성을 갖추었기 때문에 데이터에서 정보를 얻어 적용할 기회가 풍부합니다. 하지만 이런 기회를 찾아내서 대규모로 실행하려면 적절한 도구가 필요합니다. ETL 스트림 처리와 머신 러닝을 위한 엔진이 포함된 최신 클러스터 컴퓨팅 프레임워크인 Apache Spark가 바로 그런 도구입니다. 게다가 얼마 전 Databricks가 개발한 데이터 엔지니어링 기술 덕분에 Spark를 시작하기가 훨씬 쉬워졌고 IDE와 배포 파이프라인에 모두 적합한 플랫폼을 제공하게 되었습니다. Databricks 사용을 시작하자마자 새로운 수준의 데이터 문제를 해결할 준비를 하게 되었습니다.




## 사용 사례: 추천 엔진

Databricks에서 개발한 첫 번째 프로젝트는 예측 모델링을 사용하여 특정 이메일 채널을 통해 회원에게 보내는 제품 추천을 최적화하는 것이었습니다. 회원들은 구독 상자를 배송하는 주에 일련의 이메일을 받습니다. 이 이메일은 새로운 배송에 대해 알리는 한편, 상자에 추가할 수 있는 제품도 추천합니다. 회원은 클릭 몇 번만으로 이메일에서 추천 제품을 담을 수 있습니다. 우리 목표는 특정 회원에게 보내는 월간 이메일에서 어떤 제품을 어떤 순서로 홍보할지 나와 있는 제품 순위를 작성하는 것이었습니다.

전체적 탐색을 통해 회원이 각 제품에 얼마나 관심이 있는지 알 수 있는 행동을 알아냈습니다. 십여 개의 회원 데이터 세그먼트에서 여러 가지 지표를 추출하고, 수백 개의 카테고리, 작업, 태그로 해당 데이터를 분류한 다음, 불연속 시간으로 이벤트 관련 지표를 색인했습니다. 모두 합쳐서 탐색 범위 내에 있는 대규모 회원 코호트에 10,000개에 가까운 feature를 포함했습니다. 규모가 크고 희박한 고차원적 데이터 세트와 경쟁하기 위해 Spark Core, Spark SQL, MLlib를 사용하여 필수적인 ETL과 데이터 마이닝 기술을 자동화하기로 했습니다. 최종 결과물은 프로덕션 데이터로 훈련하고 튜닝한 선형 모델의 모음으로, 이를 결합하면 제품 순위를 작성할 수 있게 될 것입니다.

다음의 단계에 따라 Spark에서 완전 자동 파이프라인을 개발하기 시작했습니다.

1. 웨어하우스(Redshift)에서 데이터 추출
2. 회원당 데이터 집계 및 피벗
3. 최종 모델에 포함할 feature 선택

## 1단계: 데이터 추출

우리는 먼저 관계형 데이터베이스에서 여러 데이터 세그먼트를 살펴보는 것으로 시작했습니다. 기록 그룹을 함께 연결해서 이벤트와 관계의 영역을 설명해야 합니다. 각 데이터 세그먼트를 이해함으로써 이를 어떻게 해석하고 정제해야 할지 파악한 후 정확하고 이동할 수 있는 자기 서술적 데이터 표현을 추출해야 합니다. 데이터와 그 수명 주기에 대한 영역 전문성과 제도적 지식을 수집하는 중요한 작업이므로, 이 과정에서 알아낸 정보를 기록하고 전달하는 것이 중요합니다. Databricks는 Spark 쉘에 데이터와의 상호작용이 편리하면서도 Spark 프로그래밍 모델을 완전히 사용할 수 있는 "노트북" 인터페이스를 제공합니다. Spark 노트북은 아이디어를 시도하고 결과를 신속하게 공유하거나, 나중에 참고할 수 있도록 작업 과정을 보관하기에 좋습니다.

각 데이터 세그먼트에서 기록을 정제하고 비정규화하기 위한 정보를 추출기 모듈에 캡슐화합니다. 대부분은 Redshift에서 테이블을 내보내기만 하면 SQL 쿼리가 동적으로 생성되고 나머지 복잡한 작업은 Spark SQL에서 처리합니다. 필요한 경우, Spark의 DataFrames API를 사용하여 기능적 프로그래밍을 깨끗하게 도입할 수 있습니다. 추출기에서 기본적으로 영역별 메타데이터를 적용합니다. 특히, 특정 데이터 세그먼트를 처리하는 첫 단계는 다른 세그먼트나 파이프라인의 다른 단계와 깔끔하게 분리되어 있습니다. 추출기는 독립적으로 개발하여 테스트할 수 있습니다. 다른 탐색 작업이나 프로덕션 파이프라인에 재사용할 수도 있습니다.

```
def performExtraction(
extractorClass, exportName, joinTable=None, joinKeyCol=None,
startCol=None, includeStartCol=True, eventStartDate=None
):
customerIdCol = extractorClass.customerIdCol
timestampCol = extractorClass.timestampCol
extrArgs = extractorArgs(
customerIdCol, timestampCol, joinTable, joinKeyCol,
startCol, includeStartCol, eventStartDate
)
Extractor = extractorClass(**extrArgs)
exportPath = redshiftExportPath(exportName)
return extractor.exportFromRedshift(exportPath)
```

**데이터 추출 파이프라인의 예시 코드.** 파이프라인은 여러 추출기 클래스에서 구현하는 인터페이스를 사용하고, 인수를 전달해 동작을 직접 정의합니다. 각 추출의 세부적 과정과는 무관하게 동작합니다.

```
def exportFromRedshift(self, path):
export = self.exportDataFrame()
writeParquetWithRetry(export, path)
return sqlContext.read.parquet(path)
.persist(StorageLevel.MEMORY_AND_DISK)

def exportDataFrame(self):
self.registerTempTables()
query = self.generateQuery()
```

**추출기 코드의 예시 코드.** 대부분의 경우 추출기는 SQL 쿼리를 생성해서 SparkSQL에 전달합니다.

## 2단계: 집계 및 피벗

웨어하우스에서 추출된 데이터는 각 이벤트와 관계에 대해 거의 모든 정보가 나와 있습니다. 하지만 실제로 필요한 정보는 시간에 따른 활동의 집계 정보이므로, 어떤 제품이나 다른 제품에 대한 관심을 나타내는 행동을 효과적으로 검색할 수 있습니다. 특정 이벤트 유형을 하나씩 비교하는 것은 이 정도의 세분화 수준에서는 효과적이지 못합니다. 데이터 밀도가 매우 낮기 때문에 머신 러닝의 좋은 재료가 될 수 없습니다. 우선, 불연속적 기간에 대해 이벤트 관련 데이터를 집계해야 합니다. 이벤트를 개수, 합계, 평균, 빈도 등으로 압축하면 회원을 비교하는 데 효과적이고 데이터 마이닝을 다루기 쉬워집니다. 물론, 시간뿐만 아니라 여러 가지 차원에 대해 동일한 이벤트 세트를 집계할 수도 있습니다. 이 모든 수치는 저마다 흥미로운 이야기를 품고 있습니다.

데이터 세트에서 여러 가지 속성 각각에 대해 집계하는 작업을 데이터 피벗(롤업)이라고 합니다. 회원을 기준으로 데이터를 그룹화하고, 시간과 다른 관심 속성으로 피벗하면, 개별 이벤트와 관계에 대한 데이터가 우리 회원을 설명하는 (매우 긴) feature 목록이 됩니다. 각 데이터 세그먼트에 대해 데이터를 의미 있는 방식으로 피벗하기 위한 메서드를 자체적인 모듈에 캡슐화합니다. 이를 '모듈 변환기'라고 합니다. 피벗된 데이터 세트는 매우 광범위할 수 있으므로, DataFrames보다는 RDD로 작업하는 편이 적절합니다. 일반적으로 조밀하지 않은 벡터 형식을 사용하여 피벗된 feature set을 표현하고, 키-값 RDD 변환을 사용하여 데이터를 압축합니다. 회원의 행동을 조밀하지 않은 벡터로 표현하면 메모리에서 데이터 세트의 용량이 줄어들고, 파이프라인의 다음 단계에서 MLlib에 사용할 훈련 세트를 손쉽게 생성할 수 있습니다.

## 3단계: 데이터 마이닝

파이프라인에서 이 시점이 되면 각 회원에 대해 매우 다양한 feature set이 준비됩니다. 이제 각 제품에 대해 feature들의 어떤 하위 집합이 회원의 제품 구매 관심사를 가장 잘 나타내는지 확인해야 합니다. 이는 데이터 마이닝 문제입니다. 고려해야 할 feature가 많지 않을 경우(수천 개가 아니라 수십 개 정도) 처리할 방법은 여러 가지가 있습니다. 그러나 수없이 많은 feature를 고려할 때가 특히 어렵습니다. Databricks 플랫폼 덕분에 문제를 해결하는 데 엄청난 양의 컴퓨팅 시간을 쉽게 투자할 수 있었습니다. 우리는 비교적 규모가 작고 무작위로 샘플을 추출한 feature set에서 모델을 훈련하고 평가하는 방법을 사용했습니다. 수백 번의 반복 개선을 거쳐서 점진적으로 feature들의 하위 집합을 누적하였고, 이들 각각은 고성능 모델을 구축하는 데 중요한 기여를 합니다. 모델을 훈련하고 그 모델에서 각 feature에 대한 평가 통계를 계산하려면 컴퓨팅 리소스가 많이 소모됩니다. 하지만 대규모 Spark 클러스터를 간편하게 확장해서 각 제품에 대한 작업에 적용하였고, 작업이 종료되면 클러스터도 닫았습니다.

우리에게는 데이터 마이닝의 진행 과정을 모니터링할 수 있는 수단이 꼭 필요했습니다. 최적 성능의 모델로 수렴하는 과정에 방해가 되는 버그나 데이터 품질 문제가 있을 경우, 최대한 조기에 발견해서 처리 시간을 낭비하지 않도록 해야 합니다. 이를 위해서 Databricks에 각 반복 개선에서 수집한 평가 통계를 시각화하는 간단한 대시보드를 개발했습니다.

## 최종 모델

MLlib의 평가 모듈은 아주 간단하게 모델의 매개변수를 튜닝할 수 있습니다. ETL과 데이터 마이닝을 적용하는 고된 작업이 끝나면 최종 모델은 거의 순식간에 생성됩니다. 최종 모델 계수와 매개변수를 확인한 후, 프로덕션에서 제품 순위를 작성할 수 있습니다. 우리는 Databricks의 스케줄링 기능을 사용하여 일별로 작업을 실행하고, 그날 이메일 알림을 받을 각 회원에 대해 제품 순위를 생성했습니다. 각 회원에 대해 feature vector를 생성하기 위해 원본 훈련 데이터를 생성했던 것과 동일한 추출기와 변환기 모듈에 가장 최신 데이터를 적용했습니다. 이렇게 하면 미리 개발 시간을 절약할 수 있을 뿐만 아니라, 탐색과 프로덕션 파이프라인을 이중적으로 관리해야 하는 부담도 덜 수 있습니다. 또한, 가장 유리한 조건(훈련 데이터와 정확히 동일한 의미와 컨텍스트를 지닌 feature)으로 모델을 적용할 수 있습니다.

## Databricks 및 Apache Spark를 사용한 향후 계획

제품 추천 프로젝트가 큰 성공을 거둔 덕분에 유사한 규모의 데이터 프로젝트를 시도할 용기를 얻었습니다. Databricks는 앞으로도 우리 개발 워크플로, 특히 데이터 제품을 지원하는 데 중요한 역할을 할 것입니다. 대규모 데이터 마이닝은 전략적으로 중요한 질문에 대한 정보를 수집하는 데 필수적인 도구가 되었고, 이를 통해 얻은 예측 모델은 프로덕션에서 지능적인 feature에 배포할 수 있습니다. 머신 러닝 외에도 Spark Streaming을 기반으로 한 스트림 처리 애플리케이션을 사용하는 프로젝트도 있습니다. 예를 들어, 다양한 이벤트 스트림을 사용하여 적절히 지표를 수집하고 보고하거나, 실시간에 가깝게 시스템 전체에 데이터를 복제합니다. 물론, Spark에서도 여러 가지 ETL 프로세스를 개발하고 있습니다.

10장:

# Databricks에서 Apache SparkR로 대규모 시뮬레이션 병렬화

글: Wayne W. Jones, Dennis Vallinga 및 Hossein Falaki

2017년 6월 23일

## 소개

Apache Spark 2.0은 기존 R 함수를 병렬화할 수 있는 Apache Spark의 R 인터페이스인 SparkR에 새로운 API들을 도입했습니다. 새로운 dapply, gapply, spark.lapply 메서드는 R 사용자에게 좋은 기회를 열어주었습니다. 이 게시물에서는 Shell Oil Company 및 Databricks가 공동으로 개발한 사용 사례를 자세히 소개하고자 합니다.

## 사용 사례: 재고 추천

Shell의 기존 재고 보관 방식은 공급업체 추천, 이전 운영 경험, "직감"에 의존하는 경우가 많았습니다. 그래서 이러한 결정에 과거 데이터를 반영하는 데는 생각이 미치지 못했고, Shell 시설(예: 유정)에 재고가 과도하게 쌓이거나 부족한 경우가 생겼습니다.

Shell은 Inventory Optimization Analytics 솔루션이라는 프로토타입 도구 덕분에 SAP 재고 데이터에 다음과 같은 고급 데이터 분석 기술을 적용할 수 있었습니다.

- 창고 재고 수준 최적화
- 안전 재고 수준 예측
- 느리게 움직이는 재료 합리화
- 재료 목록에서 비재고, 재고 품목을 검토하고 재할당
- 재료 중요도 식별(예: 재료표 연결, 과거 사용량, 리드 타임)

재료에 필요한 권장 재고 보관 수준을 계산하기 위해 데이터 사이언스팀은 Markov Chain Monte Carlo(MCMC) 부트스트래핑 통계 모델을 R로 구현했습니다. 이 모델은 모든 50개 이상 Shell 시설에서 사용하는 모든 재료(일반적으로 3,000개 이상)에 적용되었습니다. 각 재료 모델은 MCMC를 1만 회 반복 개선하여 과거의 문제 분포를 나타냅니다. 모두 합치면 컴퓨팅 작업이 만만치 않게 크지만, 다행히 당황스러울 정도로 병렬적 성격이 강해 모델을 각 재료에 독립적으로 적용할 수 있었습니다.

## 기존 설정

현재 전체 모델은 48코어 192GB RAM 독립적 물리적 사무용 PC에서 실행됩니다. MCMC 부트스트랩 모델은 여러 가지 타사 R 패키지를 사용하는 맞춤형 함수 세트입니다

```
(“fExtremes”, “ismev”, “dplyr”, “tidyr”, “stringr”)
```

이 스크립트는 각 Shell 시설에 대해 반복하면서 48개 코어에서 과거에 사용했던 재료를 대략 동일한 규모의 재료 그룹으로 나눕니다. 그러면 각 코어가 모델을 개별 재료에 반복적으로 적용합니다. 우리는 재료를 그룹화하기로 했습니다. 재료에 대한 단순 루프는 각 계산이 2~5초가 소요되므로 너무 많은 오버헤드(예: R 프로세스 시작)를 발생시키기 때문입니다. 각 코어에서 재료 그룹 작업의 분산은 R **병렬 패키지**를 통해 구현합니다. 각 48개 코어 작업이 마지막까지 완료되면 스크립트는 다음 위치로 옮겨서 이 과정을 반복합니다. 모든 Shell 시설에 대한 추천 재고 수준을 계산하는 데 약 48시간이 걸렸습니다.

## Databricks에서 Apache Spark 사용

**코어가 여러 개인 하나의 대규모 컴퓨터를 사용하는 대신 클러스터 컴퓨팅을 확장하기로 했습니다. 이 사용 사례에는 Apache Spark의 새로운 R API가 적합했습니다. SparkR의 확장성과 성능을 확인하기 위해 두 버전의 워크로드가 프로토타입으로 개발되었습니다.**

## 프로토타입 I: 개념 증명

첫 프로토타입에서는 코드 변경을 최소화하려고 노력했습니다. 새로운 SparkR API가 워크로드를 처리할 수 있는지 간단하게 검증하는 것이 목표였기 때문입니다. 모든 변경 사항은 다음과 같은 시뮬레이션 단계로 제한했습니다.

각 Shell 시설 목록 요소:

1. 입력 날짜를 Spark DataFrame으로 병렬화
2. `SparkR::gapply()`를 사용하여 각 청크에 대한 병렬 시뮬레이션을 실행합니다.

기존 시뮬레이션 코드베이스에 대한 변경은 제한한 채로, Databricks의 50노드 Spark 클러스터에서 시뮬레이션 시간 합계를 3.97시간으로 단축했습니다.

## 프로토타입 II: 성능 개선

첫 번째 프로토타입은 빠르게 구현할 수 있었지만 한 가지 성능 병목이 심각했습니다. 시뮬레이션을 반복할 때마다 Spark 작업이 실행된다는 문제가 있었습니다. 이 데이터는 매우 왜곡이 심해서, 매 작업에서 실행기가 스트래글러의 종료를 기다렸다가 다음 작업에서 투입되어야 했습니다. 게다가 각 작업을 시작할 때 데이터를 Spark DataFrame으로 병렬화하는 데 시간을 소모했고 클러스터의 CPU 코어는 대부분 유휴 상태가 되었습니다.

이 문제를 해결하기 위해 전처리 단계를 수정하여 모든 위치와 재료 값에 대해 미리 입력값과 보조 날짜를 생성하도록 했습니다. 입력 날짜는 대규모 Spark DataFrame로 병렬화되었습니다. 그다음에는 두 개의 키(위치 ID, 재료 ID)로 하나의 `SparkR::gapply()`를 호출하여 시뮬레이션을 실행했습니다.

이런 간단한 개선만으로 Databricks의 50노드 Spark 클러스터에서 시뮬레이션 시간을 45분으로 단축할 수 있었습니다.

## SparkR 개선

SparkR은 Apache Spark에 가장 최근에 추가된 버전이고, 이 글을 작성하던 시점에 apply API 패밀리가 가장 최근에 SparkR에 추가되었습니다. 이 실험을 진행하는 동안 SparkR에서 여러 가지 한계와 버그를 발견하였고 Apache Spark에 신고했습니다.

- [SPARK-17790] 2GB 이상 R data.frame 병렬화 지원
- [SPARK-17919] SparkR에서 RBackend 구성에 시간 초과 적용
- [SPARK-17811] SparkR에서 Date 열에 NA 또는 NULL이 있으면 data.frame 병렬화 불가

## 다음 단계

여러분이 SparkR 개발자이고 SparkR을 살펴보고 싶다면 Databricks에 가입하고 SparkR 문서를 참고하세요.

11장: 고객 사례 분석

# Comcast는 엔터테인먼트의 미래를 제공합니다


"Databricks를 사용하면 더욱 정보에 입각한 결정을, 더욱 빠르게 내릴 수 있습니다."

— Jim Forsythe
이사, Comcast 제품 분석 및 행동 과학 부문

Comcast는 수백만 명의 고객에게 개인화된 경험을 연결해주는 글로벌 기술 미디어 기업이지만, 어마어마하게 많은 데이터, 취약한 데이터 파이프라인, 데이터 사이언스 협업 부족으로 인해 어려움을 겪었습니다. Delta Lake, MLflow와 함께 Databricks를 사용한 덕분에 페타바이트 규모의 데이터에 적절한 성능을 제공하는 데이터 파이프라인을 구축하였고, 수백 개의 모델 수명 주기를 간편하게 관리함으로써 음성 인식과 머신 러닝을 활용하는 혁신적이고 독창적이며 시청자 환경을 구현하였고 수상도 했습니다.

**사용 사례:** 경쟁이 치열한 엔터테인먼트 산업에는 '일시 정지' 버튼을 누를 여유는 없습니다. Comcast는 데이터 입력에서 고객이 만족할 만한 새로운 기능을 제공하는 머신 러닝 모델 배포에 이르기까지 모든 기술을 현대화해야 할 필요성을 느꼈습니다.

**솔루션 및 이점:** Comcast는 통합 분석 전략을 도입한 덕분에 미래형 AI 기반 엔터테인먼트로 나아갈 수 있었습니다. 독보적인 고객 환경으로 시청자들이 즐겁게 몰입할 수 있도록 지원하고 있습니다.

- **Emmy 상을 수상한 시청자 환경:** Databricks는 Comcast가 시청자 참여를 높여주는 지능적 음성 명령으로 매우 혁신적이고 수상 경력에 빛나는 시청자 환경을 구현하도록 도왔습니다.
- **컴퓨팅 비용 10배 절감:** Delta Lake로 데이터 수집을 최적화하고 나서 640개 컴퓨터를 64개로 줄이면서도 성능까지 개선했습니다. 개발팀은 인프라 관리에 소비하는 시간을 줄이고 분석에 더욱 집중할 수 있습니다.
- **데이터 사이언스 생산성 향상:** Delta Lake로 업그레이드하여 사용한 덕분에 하나의 인터랙티브 업무 공간에서 여러 프로그래밍 언어가 지원되어서 데이터 사이언티스트들의 전반적 협업이 강화되었습니다. 또한, 데이터팀은 데이터 파이프라인에서 언제든 데이터를 사용하고, 새로운 모델을 더욱 빠르게 구축하고 훈련할 수 있게 되었습니다.
- **모델 배포 속도 단축:** Comcast는 현대화를 통해 운영팀이 서로 다른 플랫폼에 모델을 배포하는 시간을 몇 주에서 몇 분으로 단축했습니다.

자세히 알아보기

11장: 고객 사례 분석

# Regeneron은 유전체 서열을 포함한 약물 발견을 가속화합니다


"Databricks Unified Data Analytics Platform 은 의사 겸 연구자, 컴퓨팅 생물학자에 이르기까지 통합 약물 개발 과정에 참여하는 모든 사람이 모든 데이터에서 인사이트에 쉽게 액세스하고, 분석, 추출할 수 있도록 지원합니다."

— Jeffrey Reid, Ph.D.
Regeneron 유전체 정보 책임자

Regeneron는 유전체 데이터를 활용하여 도움이 필요한 환자에게 새로운 약을 제공하는 것을 목표로 합니다. 하지만 이 데이터에서 사람들의 인생을 바꿀 만한 발견과 표적화된 치료법을 얻어내는 것이 그 어느 때보다도 어려워졌습니다. 데이터팀에서는 처리 성능이 부족하고 확장에 한계가 있어서 페타바이트 규모의 유전자 데이터와 임상 데이터를 분석할 수 없었습니다. Databricks는 모든 유전체 데이터 세트를 신속하게 분석하여 새로운 치료법을 발견하는 기간을 단축할 수 있도록 지원합니다.

**사용 사례:** 현재 약물 개발 파이프라인에 있는 모든 실험적 약물의 95% 이상이 실패할 것으로 예상됩니다. Regeneron Genetics Center는 이런 상황을 개선하기 위해 40만 명 이상의 순서가 배열된 진유전체와 전자 건강 기록을 결합해 세계에서 가장 종합적인 유전체 데이터베이스를 구축했습니다. 그러나 이 방대한 데이터 세트를 분석하는 데는 여러 가지 어려움이 따랐습니다.

- 유전체와 임상 데이터는 매우 분산되어 있어서 전체 10TB 데이터 세트에 대해 모델을 분석하고 훈련하기가 매우 어렵습니다.
- 기존 아키텍처가 800억 개의 데이터 포인트를 분석하도록 지원하기는 어려울 뿐만 아니라 비용도 많이 들어갑니다.
- 데이터팀은 분석에 사용할 수 있도록 데이터에 ETL을 적용하는 데만 며칠이 걸립니다.

**솔루션 및 이점:** Databricks는 Amazon Web Services에서 실행되는 Unified Data Analytics Platform을 제공하여 운영을 단순화하고 데이터 사이언스의 생산성을 높여 약물 발견 기간을 단축했습니다. Regeneron에서는 이전에는 사용할 수 없던 새로운 방식으로 데이터를 분석할 수 있게 되었습니다.

- **약물 타겟 식별 가속화:** 데이터 사이언티스트와 컴퓨팅 생물학자가 전체 데이터 세트에 쿼리를 실행하는 데 걸리는 시간이 30분에서 3초로 줄어, 600배나 개선되었습니다!
- **생산성 향상:** 협업이 개선되고 DevOps가 자동화되었으며, 파이프라인이 가속화된 덕분에(ETL을 3주에서 2일로 단축) 다양한 연구를 지원할 수 있게 되었습니다.

자세히 알아보기

databricks

11장: 고객 사례 분석

# Nationwide는 보험 계리 모델링으로 보험을 혁신합니다

"Databricks를 사용한 이후로 모든 데이터에 대해 더욱 빠르게 모델을 훈련할 수 있어서 정확한 보험료 예측이 가능했고 수익에도 실질적 영향을 미쳤습니다."

— Bryn Clark
Nationwide 데이터 사이언티스트

데이터 가용성이 폭발적으로 늘어나고 시장에서 경쟁이 격화되면서 보험사에서도 고객에게 저렴한 보험료를 제공하기 어려워졌습니다. Nationwide는 다운스트림 ML에서 분석해야 할 보험 기록이 수억 건에 달하자, 기존 배치 분석 프로세스가 너무 느리고 부정확해서 청구 빈도와 심각도를 예측하기 위한 인사이트를 얻는 데 한계가 있다는 것을 깨달았습니다. Databricks를 사용하고 나서 대규모로 딥러닝 모델을 적용해 더욱 정확한 보험료를 예측하였고 보험금 청구에서 수익이 늘어났습니다.

**사용 사례:** 정확한 보험료를 제공하려면 보험 청구 정보를 활용하는 것이 핵심입니다. 하지만 자주 발생하지 않고 예측 불가능한 청구의 성격상, 가변적인 보험 기록을 분석할 때의 데이터 문제를 해결하기란 만만치 않았고 부정확한 보험료가 산출되었습니다.

**솔루션 및 이점:** Nationwide는 Databricks Unified Data Analytics Platform을 사용하여 데이터 입력에서 딥러닝 모델 배포에 이르기까지 모든 분석 프로세스를 관리합니다. 이 완전 관리형 플랫폼으로 IT 운영을 단순화하고 데이터 사이언스팀에 새로운 데이터 중심적 기회를 제공했습니다.

- **대규모 데이터 처리:** 전체 데이터 파이프라인의 런타임을 34시간에서 4시간으로 단축해, 성능을 9배 향상했습니다.
- **특징화 시간 단축:** 데이터 엔지니어링을 통해 5시간 걸리던 작업을 약 20분으로 줄여서 15배 빠르게 feature를 찾아낼 수 있게 되었습니다.
- **모델 훈련 속도 단축:** 훈련 시간을 50% 단축해서 새로운 모델의 시장 출시 시간을 줄였습니다.
- **모델 평가 개선:** 모델 평가를 3시간에서 5분 미만으로 줄여서 60배 개선했습니다.

11장: 고객 사례 분석

# Condé Nast는 데이터와 AI 기반 경험으로 독자 참여를 끌어올립니다

CONDÉ NAST

"Databricks는 믿기 힘들 정도로 강력한 엔드투엔드 솔루션이 되어주었습니다. 전문성이 서로 다른, 다양한 팀원이 신속히 모여서 방대한 데이터를 활용해 실천할 수 있는 비즈니스 결정을 내릴 수 있게 되었습니다."

— **Paul Fryzel**
Condé Nast의 AI 인프라 수석 엔지니어

Condé Nast는 The New Yorker, Wired, Vogue 등 세계적으로 유명한 잡지를 보유하고 있는 세계 유수의 미디어 기업 중 하나입니다. Condé Nast는 데이터를 사용해서 인쇄물, 온라인, 동영상, 소셜 미디어를 통해 1억 명 이상과 연결합니다.

**사용 사례:** 주요 언론사인 Condé Nast는 포트폴리오에서 20개 이상 브랜드를 관리합니다. 매월 웹 사이트는 1억 명 이상 방문하고, 페이지 조회수는 8억 회 이상에 달해서 엄청난 양의 데이터가 발생합니다. 데이터팀은 머신 러닝을 사용하여 사용자 참여를 개선하고, 개인화된 콘텐츠 추천과 타겟 광고를 제공하는 데 집중합니다.

**솔루션 및 이점:** Databricks는 Condé Nast에 운영을 단순화하고, 우수한 성능을 제공하면서도 데이터 사이언스 혁신을 지원하는 완전 관리형 클라우드 플랫폼을 제공합니다.

- **고객 참여 개선:** Condé Nast는 데이터 파이프라인을 개선하자 더욱 빠르고 정확하고 유익하게 콘텐츠 추천을 통해 사용자 환경도 개선했습니다.
- **확장할 수 있는 설계:** 규모와 관계없이 데이터 세트를 처리하고 인사이트를 얻을 수 있습니다.
- **모델의 프로덕션 배포 증가:** MLflow를 사용한 이후로 데이터 사이언스팀은 제품을 더욱 빠르게 혁신할 수 있습니다. 1,200개 이상의 모델을 프로덕션에 배포했습니다.

자세히 알아보기

databricks

11장: 고객 사례 분석

# Showtime은 ML을 이용한 데이터 기반 콘텐츠 프로그래밍을 제공합니다

"Databricks 플랫폼을 사용한 덕분에 데이터 사이언티스트만으로 구성된 전담팀이 그동안 문제가 되었던 구성 문제를 모두 뒤로 하고 엄청난 도약을 할 수 있습니다. 생산성이 극적으로 개선되었습니다."

— Josh McNutt
SHOWTIME 데이터 전략 및 소비자 분석 전무

SHOWTIME®은 프리미엄 TV 네트워크 및 스트리밍 서비스이며, "Shameless," "Homeland," "Billions," "The Chi," "Ray Donovan," "SMILF," "The Affair," "Patrick Melrose," "Our Cartoon President," "Twin Peaks" 등과 같은 수상 경력에 빛나는 오리지널 시리즈와 오리지널 리미티드 시리즈를 제공합니다.

**사용 사례:** SHOWTIME의 데이터 사이언스팀은 조직 전체에서 데이터와 분석을 민주화하는 데 초점을 맞춥니다. SHOWTIME은 방대한 구독자 데이터(예: 시청한 방송, 시간대, 사용한 기기, 구독 기록)를 수집하여 머신 러닝으로 구독자의 행동을 예측하고, 방송 편성과 프로그래밍을 개선합니다.

**솔루션 및 이점:** Databricks는 SHOWTIME이 조직 전반에 데이터와 머신 러닝을 민주화하고 더욱 데이터 중심적 문화를 조성하도록 도왔습니다.

- **파이프라인 속도 6배 향상:** 24시간 이상 걸리던 데이터 파이프라인이 4시간 미만으로 실행되어, 팀에서 더욱 빠르게 결정을 내릴 수 있습니다.
- **인프라 복잡성 제거:** 클라우드에서 자동 클러스터 관리를 활용하는 완전 관리형 플랫폼을 사용한 데이터 사이언스팀은 하드웨어 구성, 클러스터 프로비저닝, 디버깅 등을 대신해 머신 러닝에 집중할 수 있습니다.
- **구독자 환경 혁신:** 데이터 사이언스 협업과 생산성을 개선한 덕분에 새로운 모델과 기능을 출시하는 기간이 단축되었습니다. 더욱 신속하게 실험하고, 구독자에게 더욱 개인화된 우수한 환경을 제공할 수 있습니다.

자세히 알아보기

11장: 고객 사례 분석

# Shell은 더 깨끗한 세상을 위한 에너지 솔루션으로 혁신합니다

"Shell은 Databricks를 통해 엄청난 가치를 창출했습니다. [Databricks 기반의] 재고 최적화 도구는 우리 조직에서 나온 최초의 확장된 디지털 제품으로, 이제 전 세계에 배포되어서 매년 수백만 달러를 절약할 수 있게 되었습니다."

— Daniel Jeavons
Shell 고급 분석 부문 혁신 센터 대표

Shell은 석유 가스 탐사 및 생산 기술로 유명한 선도적 기업으로서, 석유, 천연가스 생산, 휘발유 및 천연가스 판매, 석유화학 제품 제조 분야에서 세계 최고 수준을 자랑합니다.

**사용 사례:** Shell은 생산을 유지하기 위해 전 세계 시설에 3,000개 이상의 예비 부품을 보관합니다. 적절한 시점에 적절한 부품을 제공해야 생산이 중단되지 않지만, 비용을 늘리는 과도한 재고 축적을 방지하는 것도 그만큼 중요합니다.

**솔루션 및 이점:** Databricks는 Shell에 재고 및 공급망 관리를 개선하는 데 도움이 되는 클라우드 기반 통합 분석 플랫폼을 제공합니다.

- **예측 모델링:** 확장할 수 있는 예측 모델은 50개 이상의 위치에서 3,000개 이상의 재료 유형에 대해 개발, 배포됩니다.
- **과거 분석:** 각 재료 모델은 Markov Chain Monte Carlo를 1만 회 반복 시뮬레이션하여 과거의 문제 분포를 나타냅니다.
- **엄청난 성능 향상:** 데이터 사이언스팀은 성능 향상에 집중하여 Databricks의 50노드 Apache Spark™ 클러스터에서 재고 분석과 예측 시간을 48시간에서 45분으로 단축해, 32배나 성능을 높였습니다.
- **경비 절감:** 연간 수백만 달러에 달하는 비용이 절감됩니다.

자세히 알아보기

databricks

11장: 고객 사례 분석

# Riot Games는 AI를 이용해 게이머의 참여를 유도하고 이탈을 줄입니다

"데이터 사이언티스트들을 클러스터 관리에서 해방시켜주고 싶었습니다. 사용하기 편리한 관리형 Spark 솔루션을 Databricks에서 사용한 후로 가능하게 되었죠. 이제 우리 팀은 게임 환경을 개선하는 데 집중할 수 있습니다."

— Colin Borys
Riot Games 데이터 사이언티스트

Riot Games의 목표는 세계에서 가장 플레이어 중심적인 게임 기업이 되는 것입니다. 2006년에 로스앤젤레스에서 설립된 Riot Games는 '리그 오브 레전드' 게임으로 가장 유명합니다. 매월 게임을 플레이하는 게이머가 1억 명 이상입니다.

**사용 사례:** 네트워크 성능 모니터링을 통해 게임 환경을 개선하고 게임 내 욕설을 차단합니다.

**솔루션 및 이점:** Databricks는 확장 가능하고 빠른 분석을 제공하여 Riot Games가 플레이어의 게임 환경을 개선하도록 도왔습니다.

- **게임 내 구매 환경 개선:** 5,000억 개 이상의 데이터 포인트를 기반으로 고유한 서비스를 제공하는 추천 엔진을 신속히 구축하고 제품화했습니다. 이제 게이머들이 원하는 콘텐츠를 더욱 쉽게 찾을 수 있습니다.
- **게임 지연 완화:** 네트워크 문제를 실시간으로 탐지하는 ML 모델을 구축하였고 플레이어에게 부정적인 영향을 미치기 전에 장애를 해결할 수 있습니다.
- **분석 시간 단축:** 데이터 준비와 탐색 처리 성능이 EMR에 비해 50% 향상되어서 분석 속도가 상당히 빨라졌습니다.

자세히 알아보기

databricks

11장: 고객 사례 분석

# Eneco는 ML을 사용해 에너지 사용량과 운영 비용을 절감합니다

Quby는 에너지 사용량, 쾌적도, 가정 보안 등을 제어하는 스마트 에너지 관리 기기인 Toon을 개발한 기술 기업입니다. Quby의 스마트 기기는 유럽 전역의 가정에서 사용됩니다. 따라서 가정에서 사용하는 가전의 센서에서 수집한 페타바이트 규모의 IoT 데이터로 구성된 유럽 최대 규모의 에너지 데이터 세트를 관리합니다. Quby는 이 데이터를 사용하여 고객이 더욱 편안한 생활을 영위하면서도 개인 맞춤형 에너지 사용량 추천을 통해 에너지 소비를 절약하도록 지원하고자 합니다.

**사용 사례:** 개인 맞춤형 에너지 사용량 추천 머신 러닝과 IoT 데이터를 기반으로 Waste Checker 앱에서 가정 내 에너지 사용량을 줄이기 위한 개인 맞춤형 추천을 제공합니다.

**솔루션 및 이점:** Databricks는 Quby에게 Unified Data Analytics Platform을 제공하여 데이터 사이언스와 엔지니어링에서 확장 가능하고 협력적인 환경을 조성함으로써, 데이터팀이 더욱 빠르게 혁신하고 ML 기반 서비스를 Quby의 고객에게 제공할 수 있도록 했습니다.

- **저렴한 비용:** Databricks가 제공하는 비용 절감 기능(예: 자동 확장 클러스터, Spot 인스턴스) 덕분에 Quby는 인프라 관리 운영 비용을 상당히 절감하면서도 대량의 데이터를 처리할 수 있게 되었습니다.
- **빠른 혁신:** 기존 아키텍처로는 개념 증명에서 프로덕션까지 12개월 이상 걸렸습니다. Databricks를 사용한 이후로는 같은 프로세스가 8주 이내로 끝납니다. Quby의 데이터팀은 고객을 위한 새로운 ML 기반 기능을 더욱 빠르게 개발할 수 있게 되었습니다.
- **에너지 소비량 감소:** Quby는 Waste Checker 앱을 통해 개인 맞춤형 추천으로 절약한 에너지가 시간당 6,700백만 kW가 넘는다는 것을 발견했습니다.

**"Databricks는 Delta Lake와 Structured Streaming을 활용하여 매우 빠르게 고객에게 알림과 추천을 제공할 수 있도록 도와주었습니다. 고객들은 가정에서 불편해지기 전에 미리 문제를 해결하고 수정할 수 있습니다."**

— Stephen Galsworthy
Quby 데이터 사이언스 책임자

자세히 알아보기

databricks

## Databricks 소개

Databricks는 데이터 및 AI 회사입니다. Comcast, Condé Nast, Nationwide, H&M을 비롯하여 전 세계적으로 수천 개의 고객사가 Databricks의 개방적인 통합 플랫폼을 데이터 엔지니어링, 머신 러닝, 분석에 사용합니다. Databricks는 샌프란시스코에 본사가 있으며 전 세계에 지사를 두고 있는 벤처입니다. Apache Spark™, Delta Lake 및 MLflow의 창설자가 설립한 Databricks는 데이터팀이 세계적 난제를 해결하는 것을 돕고자 합니다.

Databricks에 대한 자세한 정보를 확인하려면 팔로우하세요.

Twitter, LinkedIn, Facebook

Databricks 체험하기

문의하기

databricks