eBook

The Big Book of Data Science Use Cases

A collection of technical blogs, including code samples and notebooks

databricks

Contents

CHAPTER 1:

Introduction

The world of data science is evolving so fast that it's not easy to find real-world use cases that are relevant to what you're working on. That's why we've collected together these blogs from industry thought leaders with practical use cases you can put to work right now. This how-to reference guide provides everything you need — including code samples — so you can get your hands dirty working with the Databricks platform.

CHAPTER 2:

Democratizing Financial Time Series Analysis With Databricks

Faster development with Databricks Connect and Koalas

By **Ricardo Portilla**

October 9, 2019

Try this notebook in Databricks →

Introduction

The role of data scientists, data engineers and analysts at financial institutions includes (but is not limited to) protecting *hundreds of billions of dollars'* worth of assets and protecting investors from *trillion-dollar* impacts, say, from a flash crash. One of the biggest technical challenges underlying these problems is scaling time series manipulation. Tick data, alternative data sets such as geospatial or transactional data, and fundamental economic data are examples of the rich data sources available to financial institutions, all of which are naturally indexed by timestamp. Solving business problems in finance such as risk, fraud and compliance ultimately rests on being able to aggregate and analyze thousands of time series in parallel. Older technologies, which are RDBMS-based, do not easily scale when analyzing trading strategies or conducting regulatory analyses over years of historical data. Moreover, many existing time series technologies use specialized languages instead of standard SQL or Python-based APIs.

Fortunately, Apache Spark™ contains plenty of built-in functionality such as windowing, which naturally parallelizes time series operations. Moreover, Koalas, an open source project that allows you to execute distributed machine learning queries via Apache Spark using the familiar pandas syntax, helps extend this power to data scientists and analysts.

In this blog, we will show how to build time series functions on hundreds of thousands of tickers in parallel. Next, we demonstrate how to modularize functions in a local IDE and create rich time series feature sets with Databricks Connect. Lastly, if you are a pandas user looking to scale data preparation that feeds into financial anomaly detection or other statistical analyses, we use a market manipulation example to show how Koalas makes scaling transparent to the typical data science workflow.

Set up time series data sources

Let's begin by ingesting a couple of traditional financial time series data sets: trades and quotes. We have simulated the data sets for this blog, which are modeled on data received from a trade reporting facility (trades) and the National Best Bid Offer (NBBO) feed (from an exchange such as the NYSE). You can find some example data here.

This article generally assumes basic definitions of financial terms; for more extensive references, see Investopedia's documentation. What is notable from the data sets below is that we've assigned the `TimestampType` to each timestamp, so the trade execution time and quote change time have been renamed to `event_ts` for normalization purposes. In addition, as shown in the full notebook attached in this article, we ultimately convert these data sets to Delta format so that we ensure data quality and keep a columnar format, which is most efficient for the type of interactive queries we have below.

```
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())
 ])
```

```
1 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.

```
1 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.

Merging and aggregating time series with Apache Spark

There are over 600,000 publicly traded securities globally today in financial markets. Given that our trade and quote data sets span this volume of securities, we'll need a tool that scales easily. Because Apache Spark offers a simple API for ETL and it is the standard engine for parallelization, it is our go-to tool for merging and aggregating standard metrics, which in turn help us understand liquidity, risk and fraud. We'll start with the merging of trades and quotes, then aggregate the trades data set to show simple ways to slice the data. Lastly, we'll show how to package this code up into classes for faster iterative development with Databricks Connect. The full code used for the metrics on the following page is in the attached notebook.

As-of joins

An as-of join is a commonly used "merge" technique that returns the latest right value effective at the time of the left timestamp. For most time series analyses, multiple types of time series are joined together on the symbol to understand the state of one time series (e.g., NBBO) at a particular time present in another time series (e.g., trades). The example below records the state of the NBBO for every trade for all symbols. As seen in the figure below, we have started off with an initial base time series (trades) and merged the NBBO data set so that each timestamp has the latest bid and offer recorded "as of the time of the trade." Once we know the latest bid and offer, we can compute the difference (known as the spread) to understand at what points the liquidity may have been lower (indicated by a large spread). This kind of metric impacts how you may organize your trading strategy to boost your *alpha*.

First, let's use the built-in windowing function last to find the **last** non-null quote value after ordering by time.

```
# 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"))
```

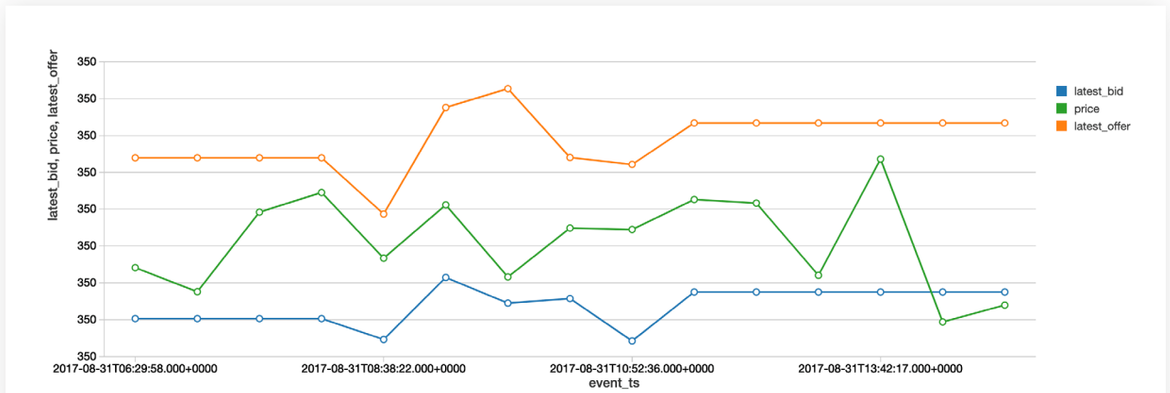
Now, we'll call our custom join to merge our data and attach our quotes. See attached notebook for full code.

```
# 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

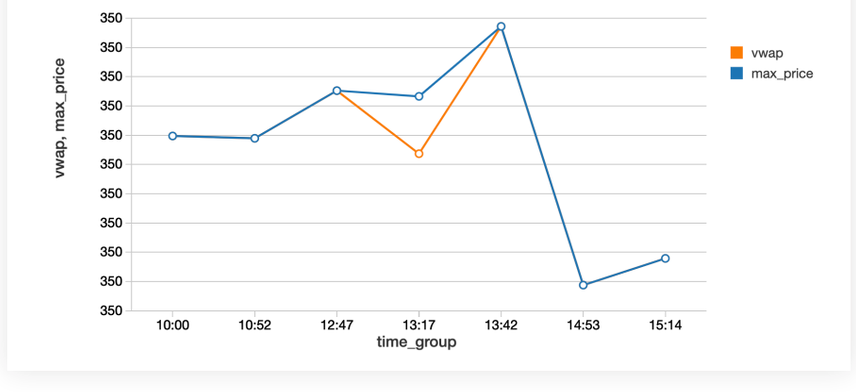
event_ts	price	symbol	ind_cd	latest_bid	latest_offer
2017-08-31T06:29:58.000+0000	347.4121586706382	K	1	346.0297772384752	350.39315623662594
2017-08-31T06:32:30.000+0000	346.7582132240916	K	1	346.0297772384752	350.39315623662594
2017-08-31T06:37:31.000+0000	348.919146315238	K	1	346.0297772384752	350.39315623662594
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



Marking VWAP against trade patterns

We've shown a merging technique above, so now let's focus on a standard aggregation, namely Volume-Weighted Average Price (VWAP), which is the average price weighted by volume. This metric is an indicator of the trend and value of the security throughout the day. The VWAP function within our wrapper class (in the attached notebook) shows where the VWAP falls above or below the trading price of the security. In particular, we can now identify the window during which the VWAP (in orange) falls below the trade price, showing that the stock is overbought.

```
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'))
```



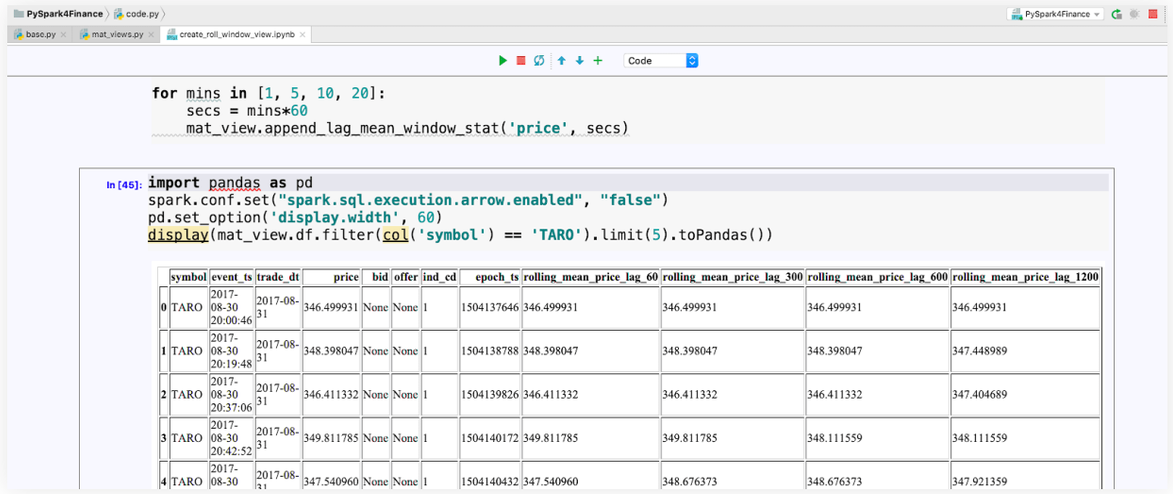
databricks

Faster iterative development with Databricks Connect

Up to this point, we've created some basic wrappers for one-off time series metrics. However, productionalization of code requires modularization and testing, and this is best accomplished in an IDE. This year, we introduced Databricks Connect, which gives the ability for local IDE development and enhances the experience with testing against a live Databricks cluster. The benefits of Databricks Connect for financial analyses include the ability to add time series features on small test data with the added flexibility to execute interactive Spark queries against years of historical tick data to validate features.

We use PyCharm to organize classes needed for wrapping PySpark functionality for generating a rich time series feature set. This IDE gives us code completion, formatting standards, and an environment to quickly test classes and methods before running code.

We can quickly debug classes then run Spark code directly from our laptop using a Jupyter notebook that loads our local classes and executes interactive queries with scalable infrastructure. The console pane shows our jobs being executed against a live cluster.



	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

Lastly, we get the best of both worlds by using our local IDE and, at the same time, appending to our materialized time series view on our largest time series data set.

Leveraging Koalas for market manipulation

The pandas API is the standard tool for data manipulation and analysis in Python and is deeply integrated into the Python data science ecosystem, e.g., NumPy, SciPy, Matplotlib. One drawback of pandas is that it does not scale easily to large amounts of data. Financial data always includes years of historical data, which is critical for risk aggregation or compliance analysis. To make this easier, we introduced Koalas as a way to leverage pandas APIs while executing Spark on the back end. Since the Koalas API matches pandas, we don't sacrifice ease of use, and migration to scalable code is a **one-line code change** (see import of Koalas in the next section). Before we showcase Koalas' fit for financial time series problems, let's start with some context on a specific problem in financial fraud: front running.

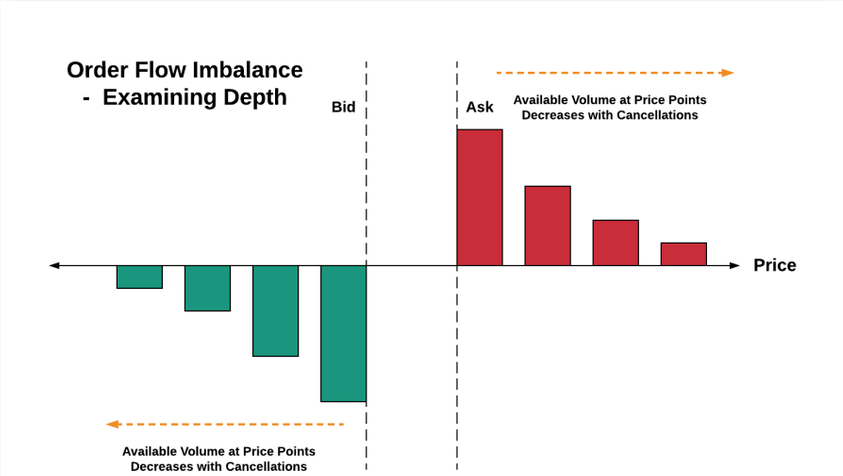
Source: CC0 Public domain images – Image on **left**, image on **right**

Front running occurs when the following sequence occurs:

1. A trading firm is aware of nonpublic information that may affect the price of a security
2. The firm buys a large bulk order (or large set of orders totaling a large aggregate volume)
3. Due to the removal of liquidity, the security price rises
4. The firm sells the security to investors (which has been driven upward from the previous purchase) and makes a large profit, forcing investors to pay a larger price even though the information upon which the security was traded was nonpublic

For illustration purposes, a simple example using farmers markets and an apple pie business is found here. This example shows Freddy, a runner who is aware of the imminent demand for apples needed for apple pie businesses across the country and subsequently purchases apples at all farmers markets. This, in effect, allows Freddy to sell his apples at a premium to buyers since Freddy caused a major impact by purchasing before any other buyers (representing investors) had a chance to buy the product.

Detection of front running requires an understanding of order flow imbalances (see diagram below). In particular, anomalies in order flow imbalance will help identify windows during which front running may be occurring.



Let's now use the Koalas package to improve our productivity while solving the market manipulation problem. Namely, we'll focus on the following to find order flow imbalance anomalies:

- De-duplication of events at the same time
- Lag windows for assessing supply/demand increases
- Merging of data frames to aggregate order flow imbalances

De-duplication of time series

Common time series data cleansing involves imputation and de-duplication. You may find duplicate values in high-frequency data (such as quote data). When there are multiple values per time with no sequence number, we need to de-duplicate so subsequent statistical analysis makes sense. In the case below, multiple bid/ask share quantities are reported per time, so for computation of order imbalance, we want to rely on one value for maximum depth per time.

```
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

Time series windowing with Koalas

We've de-duplicated our time series, so now let's look at windows so we can find supply and demand. Windowing for time series generally refers to looking at slices or intervals of time. Most trend calculations (simple moving average, for example) all use the concept of time windows to perform calculations. Koalas inherits the simple pandas interface for getting lag or lead values within a window using `shift` (analogous to Spark's lag function), as demonstrated below.

```
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

Merge on timestamp and compute imbalance with Koalas column arithmetic

Now that we have lag values computed, we want to be able to merge this data set with our original time series of quotes. Below, we employ the Koalas `merge` to accomplish this with our time index. This gives us the consolidated view we need for supply/demand computations, which lead to our order imbalance metric.

```
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

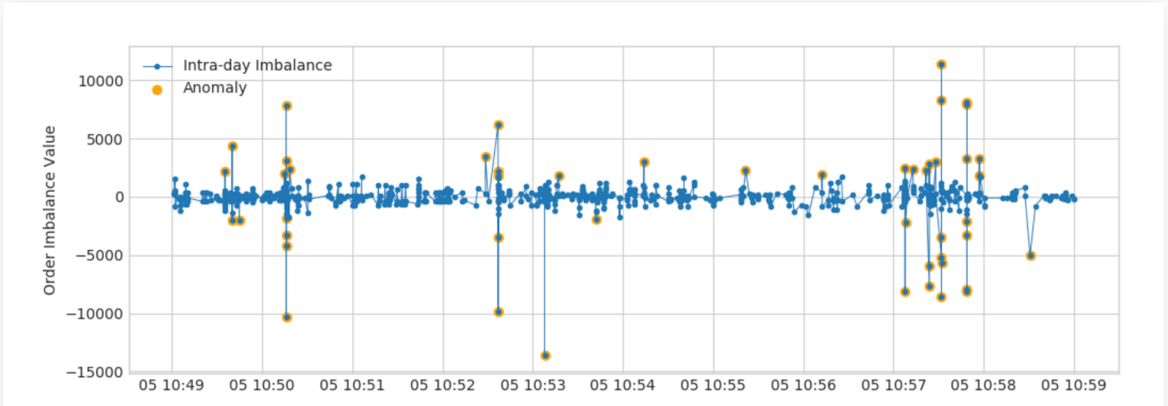
Koalas to NumPy for fitting distributions

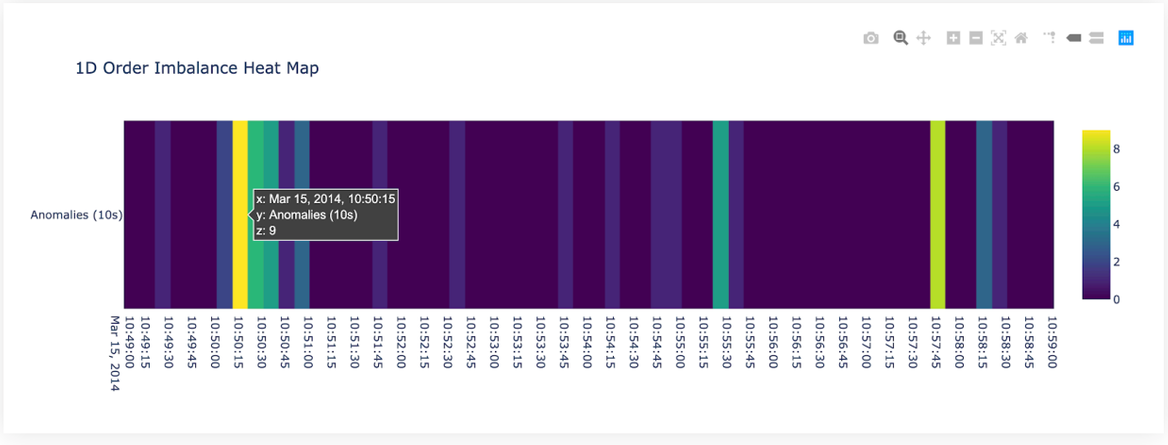
After our initial prep, it's time to convert our Koalas data frame to a format useful for statistical analysis. For this problem, we might aggregate our imbalances down to the minute or other unit of time before proceeding, but for purposes of illustration, we'll run against the full data set for our ticker "ITUB." Below, we convert our Koalas structure to a NumPy data set so we can use the SciPy library for detecting anomalies in order flow imbalance. Simply use the to_numpy() syntax to bridge this analysis.

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

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

At right, we plotted the distribution of our order flow imbalances along with markers for the 5th and 95th percentiles to identify the events during which imbalance anomalies occurred. See the full notebook for the code to fit distributions and create this plot. The time during imbalances we just computed with our Koalas/SciPy workflow will correlate with potential instances of front running, the market manipulation scheme we were searching for.

The time series visualization below pinpoints the anomalies retrieved as outliers above, highlighted in orange. In our final visualization, we use the `plotly` library to summarize time windows and frequency of anomalies in the form of a heat map. Specifically, we identify the **10:50:10–10:50:20** time frame as a potential problem area from the front running perspective.





Conclusion

In this article, we've shown how Apache Spark and Databricks can be leveraged for time series analysis both directly, by using windowing and wrappers, and indirectly, by using Koalas. Most data scientists rely on the pandas API, so Koalas helps them use pandas functionality while allowing the scale of Apache Spark. The advantages of using Spark and Koalas for time series analyses include:

- Parallelize analyses of your time series for risk, fraud or compliance use cases with as-of joins and simple aggregations
- Iterate faster and create rich time series features with Databricks Connect
- Arm your data science and quant teams with Koalas to scale out data preparation while not sacrificing pandas ease of use and APIs

Try this notebook on Databricks today! Contact us to learn more about how we assist customers with financial time series use cases.

> Start experimenting with this free Databricks notebook.

CHAPTER 3:

Understanding Dynamic Time Warping

Part 1 of our Using Dynamic Time Warping and MLflow to Detect Sales Trends series

By **Ricardo Portilla**, **Brenner Heintz** and **Denny Lee**

April 30, 2019

Try this notebook in Databricks →

Introduction

The phrase "dynamic time warping," at first read, might evoke images of Marty McFly driving his DeLorean at 88 MPH in the "Back to the Future" series. Alas, dynamic time warping does not involve time travel; instead, it's a technique used to dynamically compare time series data when the time indices between comparison data points do not sync up perfectly.

As we'll explore below, one of the most salient uses of dynamic time warping is in speech recognition — determining whether one phrase matches another, even if the phrase is spoken faster or slower than its comparison. You can imagine that this comes in handy to identify the "wake words" used to activate your Google Home or Amazon Alexa device — even if your speech is slow because you haven't yet had your daily cup(s) of coffee.

Dynamic time warping is a useful, powerful technique that can be applied across many different domains. Once you understand the concept of dynamic time warping, it's easy to see examples of its applications in daily life, and its exciting future applications. Consider the following uses:

- **Financial markets:** Comparing stock trading data over similar time frames, even if they do not match up perfectly — for example, comparing monthly trading data for February (28 days) and March (31 days)
- **Wearable fitness trackers:** More accurately calculating a walker's speed and the number of steps, even if their speed varied over time
- **Route calculation:** Calculating more accurate information about a driver's ETA, if we know something about their driving habits (for example, they drive quickly on straightaways but take more time than average to make left turns)

Data scientists, data analysts and anyone working with time series data should become familiar with this technique, given that perfectly aligned time series comparison data can be as rare to see in the wild as perfectly "tidy" data.

In this blog series, we will explore:

- The basic principles of dynamic time warping
- Running dynamic time warping on sample audio data
- Running dynamic time warping on sample sales data using MLflow

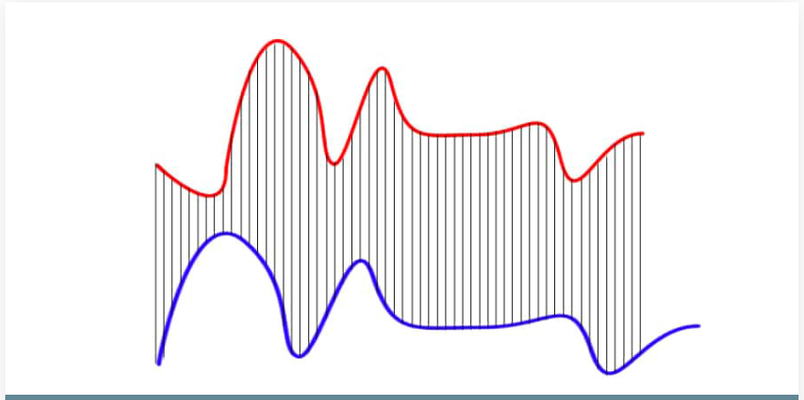
Dynamic time warping

The objective of time series comparison methods is to produce a *distance metric* between two input time series. The similarity or dissimilarity of two time series is typically calculated by converting the data into vectors and calculating the Euclidean distance between those points in vector space.

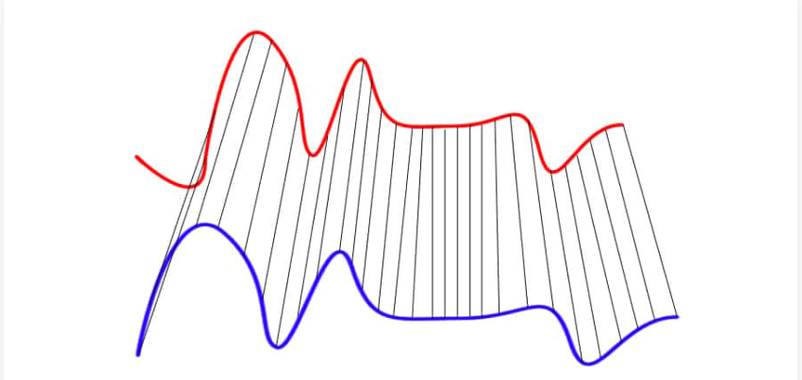
Dynamic time warping is a seminal time series comparison technique that has been used for speech and word recognition since the 1970s with sound waves as the source; an often cited paper is "Dynamic time warping for isolated word recognition based on ordered graph searching techniques."

Background

This technique can be used not only for pattern matching, but also anomaly detection (e.g., overlap time series between two disjointed time periods to understand if the shape has changed significantly or to examine outliers). For example, when looking at the red and blue lines in the following graph, note the traditional time series matching (i.e., Euclidean matching) is extremely restrictive. On the other hand, dynamic time warping allows the two curves to match up evenly even though the X-axes (i.e., time) are not necessarily in sync. Another way to think of this is as a robust dissimilarity score where a lower number means the series is more similar.



EUCLIDEAN MATCHING



DYNAMIC TIME WARP MATCHING

Source: Wikimedia Commons
File: Euclidean_vs_DTW.jpg

Two time series (the base time series and new time series) are considered similar when it is possible to map with function f(x) according to the following rules so as to match the magnitudes using an optimal (warping) path.

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

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

Sound pattern matching

Traditionally, dynamic time warping is applied to audio clips to determine the similarity of those clips. For our example, we will use four different audio clips based on two different quotes from a TV show called "The Expanse." There are four audio clips (you can listen to them below but this is not necessary) — three of them (clips 1, 2 and 4) are based on the quote:

"Doors and corners, kid. That's where they get you."

And in one clip (clip 3) is the quote:

"You walk into a room too fast, the room eats you."

Clip 1 | Doors and corners, kid. That's where they get you. [v1]

0:00 / 0:06

Clip 2 | Doors and corners, kid. That's where they get you. [v2]

0:00 / 0:08

Clip 3 | You walk into a room too fast, the room eats you.

0:00 / 0:07

Clip 4 | Doors and corners, kid. That's where they get you. [v3]

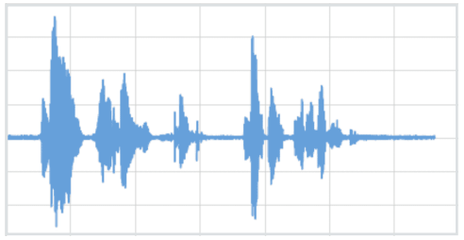
0:00 / 0:07

Quotes are from "The Expanse"

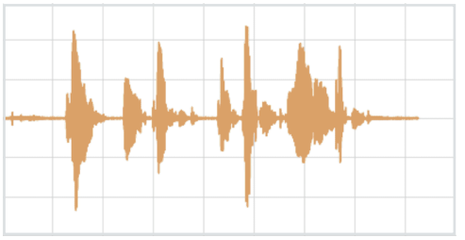
Below are visualizations using `matplotlib` of the four audio clips:

- **Clip 1:** This is our base time series based on the quote "Doors and corners, kid. That's where they get you."
- **Clip 2:** This is a new time series [v2] based on clip 1 where the intonation and speech pattern are extremely exaggerated.
- **Clip 3:** This is another time series that's based on the quote "You walk into a room too fast, the room eats you." with the same intonation and speed as clip 1.
- **Clip 4:** This is a new time series [v3] based on clip 1 where the intonation and speech pattern is similar to clip 1.

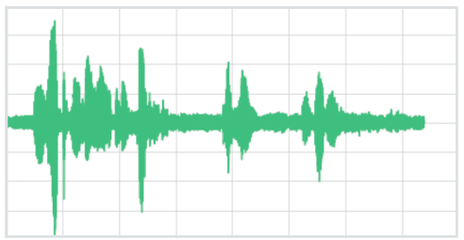
Clip 1 | Doors and corners, kid. That's where they get you. [v1]



Clip 2 | Doors and corners, kid. That's where they get you. [v2]



Clip 3 | You walk into a room too fast, the room eats you.



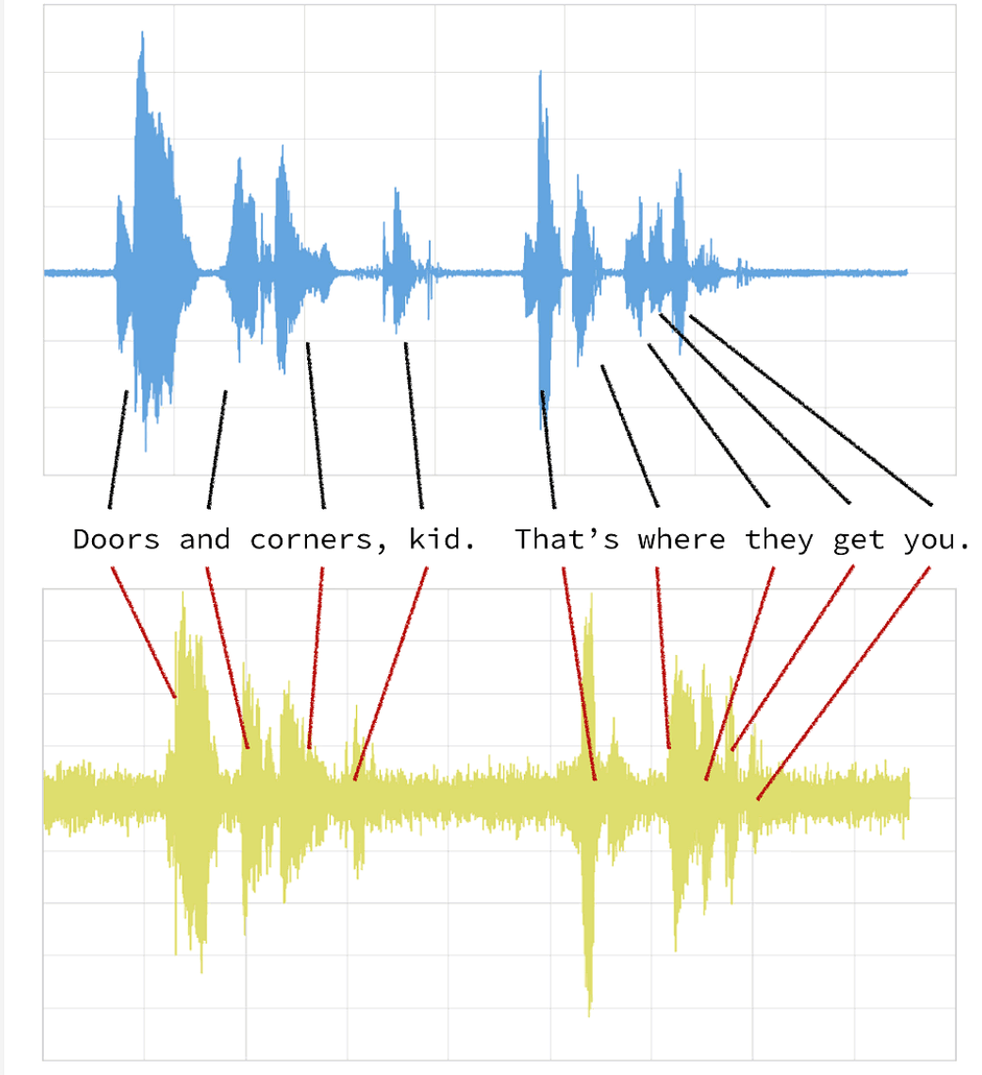
Clip 4 | Doors and corners, kid. That's where they get you. [v3]

The code to read these audio clips and visualize them using Matplotlib can be summarized in the following code snippet.

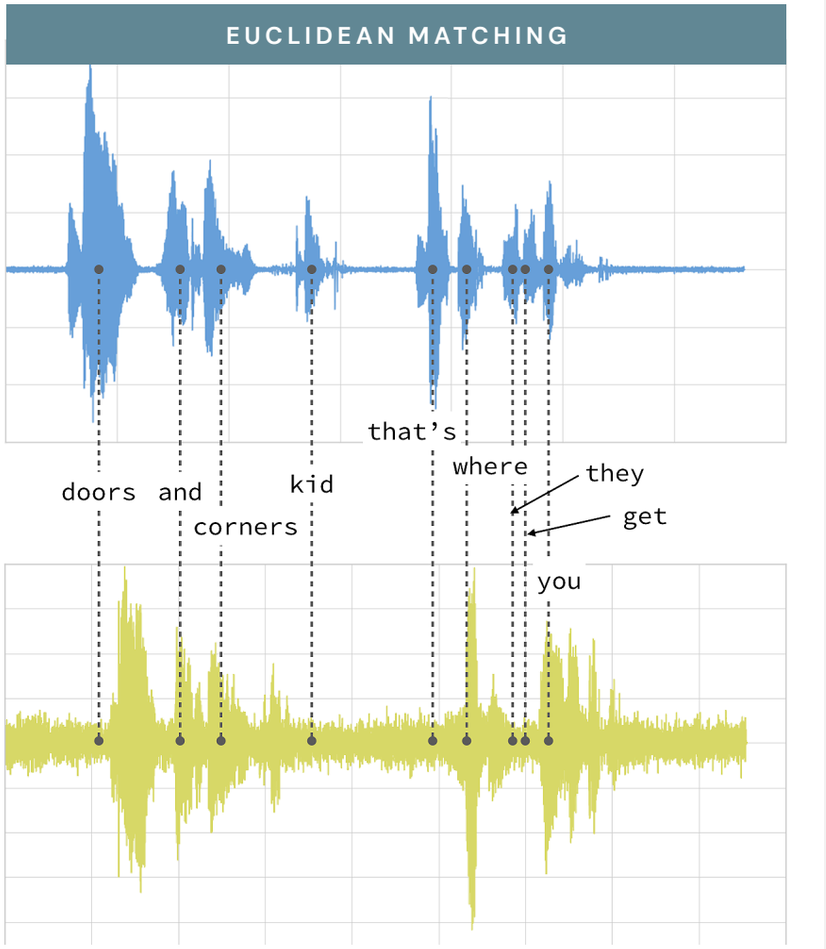
```
from scipy.io import wavfile
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure

# 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)
```

The full code base can be found in the notebook Dynamic Time Warping Background.

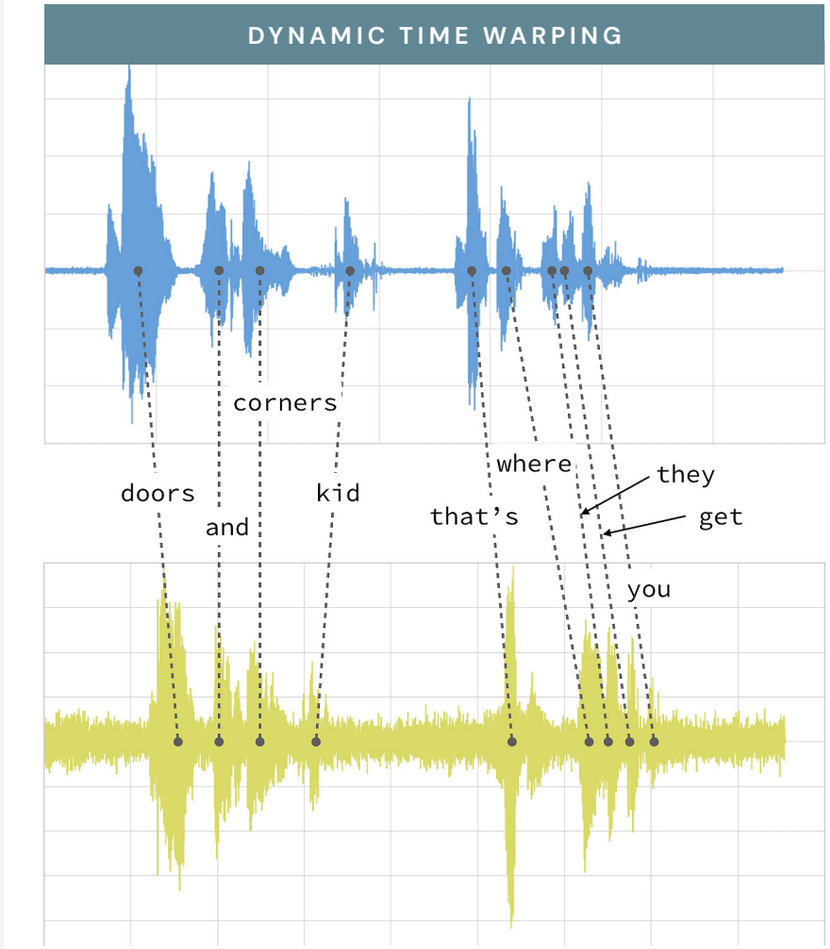
As noted below, the two clips (in this case, clips 1 and 4) have different intonations (amplitude) and latencies for the same quote.



If we were to follow a traditional Euclidean matching (per the following graph), even if we were to discount the amplitudes, the timings between the original clip (blue) and the new clip (yellow) do not match.



With dynamic time warping, we can shift time to allow for a time series comparison between these two clips.

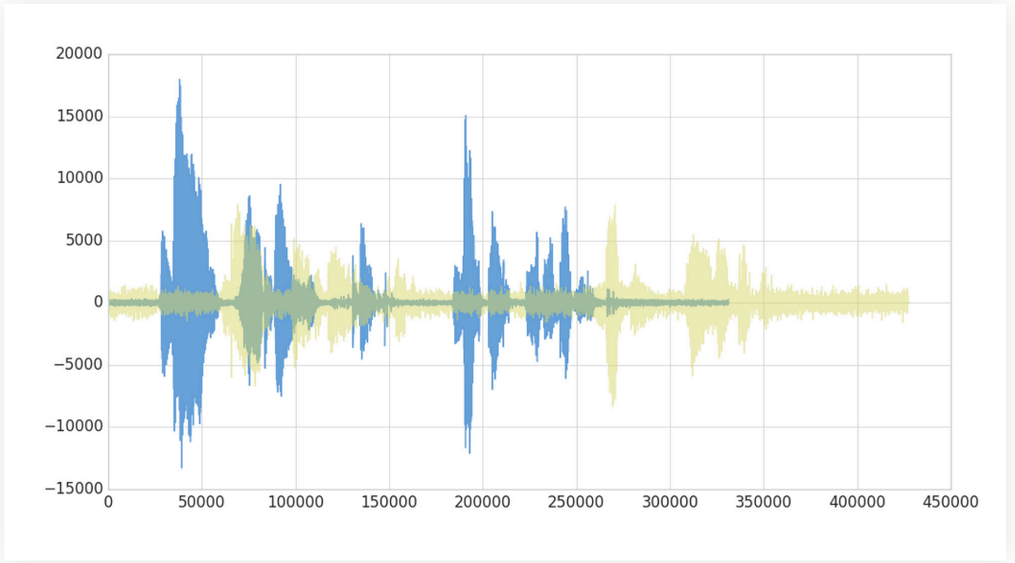


For our time series comparison, we will use the `fastdtw` PyPi library; the instructions to install PyPi libraries within your Databricks workspace can be found here: Azure | AWS. By using fastdtw, we can quickly calculate the distance between the different time series.

```
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)
```

The full code base can be found in the notebook Dynamic Time Warping Background.

BASE	QUERY	DISTANCE
Clip 1	Clip 2	480148446.0
	Clip 3	310038909.0
	Clip 4	293547478.0



Some quick observations:

- As noted in the preceding graph, clips 1 and 4 have the shortest distance, as the audio clips have the same words and intonations
- The distance between clips 1 and 3 is also quite short (though longer than when compared to clip 4) — even though they have different words, they are using the same intonation and speed
- Clips 1 and 2 have the longest distance due to the extremely exaggerated intonation and speed even though they are using the same quote

As you can see, with dynamic time warping, one can ascertain the similarity of two different time series.

Next

Now that we have discussed dynamic time warping, let's apply this use case to detect sales trends.

CHAPTER 3:

Using Dynamic Time Warping and MLflow to Detect Sales Trends

Part 2 of our Using Dynamic Time Warping and MLflow to Detect Sales Trends series

By **Ricardo Portilla**, **Brenner Heintz** and **Denny Lee**

April 30, 2019

Try this notebook series (in DBC format) in Databricks →

Background

Imagine that you own a company that creates 3D-printed products. Last year, you knew that drone propellers were showing very consistent demand, so you produced and sold those, and the year before you sold phone cases. **The new year is arriving very soon, and you're sitting down with your manufacturing team to figure out what your company should produce for next year.** Buying the 3D printers for your warehouse put you deep into debt, so you have to make sure that your printers are running at or near 100% capacity at all times in order to make the payments on them.

Since you're a wise CEO, you know that your production capacity over the next year will ebb and flow — there will be some weeks when your production capacity is higher than others. For example, your capacity might be higher during the summer (when you hire seasonal workers) and lower during the third week of every month (because of issues with the 3D printer filament supply chain). Take a look at the chart below to see your company's production capacity estimate:

databricks

Your job is to choose a product for which weekly demand meets your production capacity as closely as possible. You're looking over a catalog of products that includes last year's sales numbers for each product, and you think this year's sales will be similar.

If you choose a product with weekly demand that exceeds your production capacity, then you'll have to cancel customer orders, which isn't good for business. On the other hand, if you choose a product without enough weekly demand, you won't be able to keep your printers running at full capacity and may fail to make the debt payments.

Dynamic time warping comes into play here because sometimes supply and demand for the product you choose will be slightly out of sync. There will be some weeks when you simply don't have enough capacity to meet all of your demand, but as long as you're very close and you can make up for it by producing more products in the week or two before or after, your customers won't mind. If we limited ourselves to comparing the sales data with our production capacity using Euclidean matching, we might choose a product that didn't account for this, and leave money on the table. Instead, we'll use dynamic time warping to choose the product that's right for your company this year.

Load the product sales data set

We will use the weekly sales transaction data set found in the UCI Data Set Repository to perform our sales-based time series analysis. (Source attribution: 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))
```

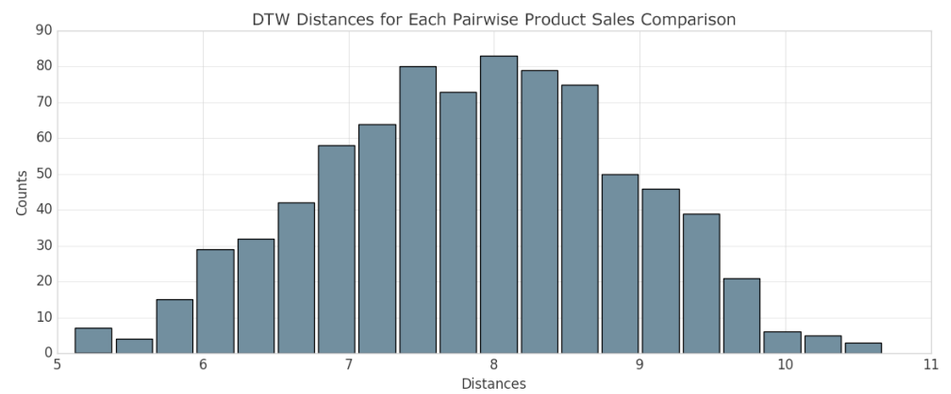
Product_Code	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
P1	11	12	10	8	13	12	14	21	6	14	11	14	16	9
P2	7	6	3	2	7	1	6	3	3	3	2	2	6	2
P3	7	11	8	9	10	8	7	13	12	6	14	9	4	7
P4	12	8	13	5	9	6	9	13	13	11	8	4	5	4
P5	8	5	13	11	6	7	9	14	9	9	11	18	8	4
P6	3	3	2	7	6	3	8	6	6	3	1	1	5	4
P7	4	8	3	7	8	7	2	3	10	3	5	2	3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

Each product is represented by a row, and each week in the year is represented by a column. Values represent the number of units of each product sold per week. There are 811 products in the data set.

Calculate distance to optimal time series by product code

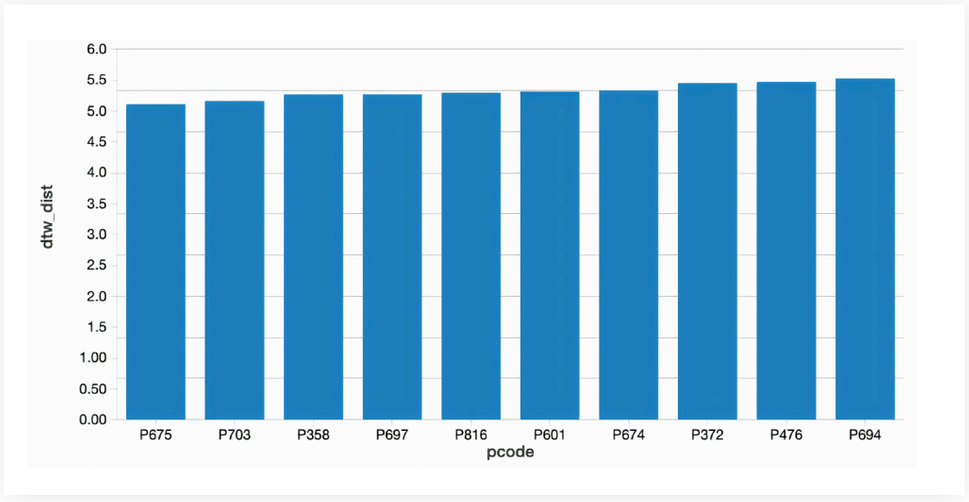
```
# Calculate distance via dynamic time warping between product
code and optimal time series
import numpy as np
import _ucrdtw

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']
```

Using the calculated dynamic time warping "distances" column, we can view the distribution of DTW distances in a histogram.

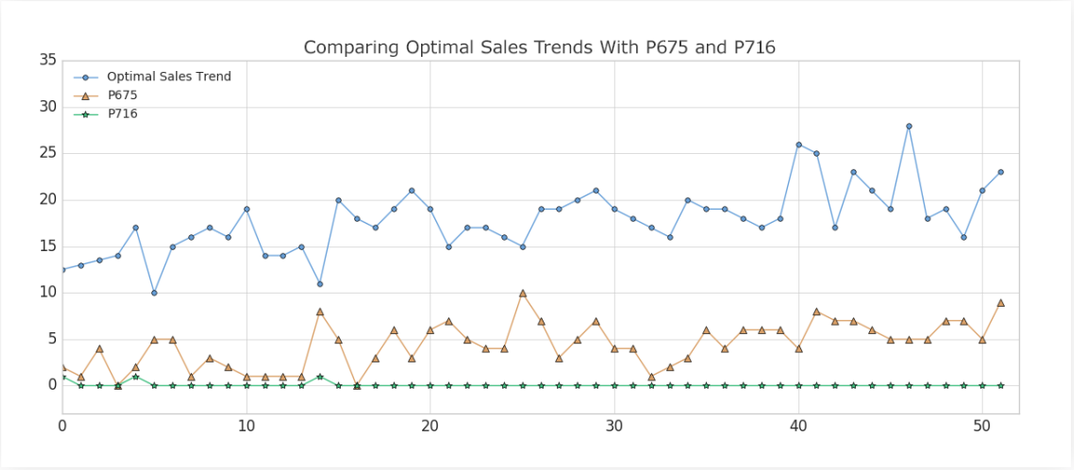


From there, we can identify the product codes closest to the optimal sales trend (i.e., those that have the smallest calculated DTW distance). Since we're using Databricks, we can easily make this selection using a SQL query. Let's display those that are closest.

```
%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
```



After running this query, along with the corresponding query for the product codes that are *furthest* from the optimal sales trend, we were able to identify the two products that are closest and furthest from the trend. Let's plot both of those products and see how they differ.

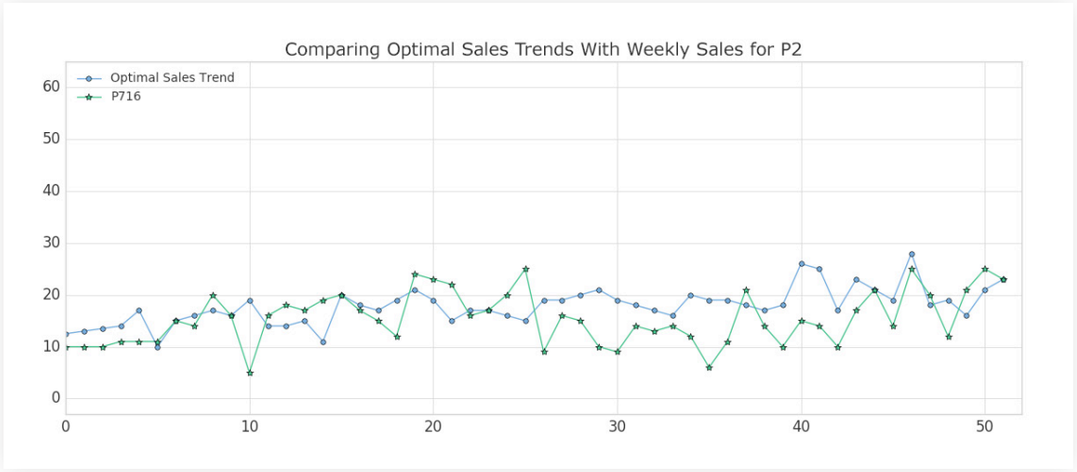


As you can see, Product #675 (shown in the orange triangles) represents the best match to the optimal sales trend, although the absolute weekly sales are lower than we'd like (we'll remedy that later). This result makes sense since we'd expect the product with the closest DTW distance to have peaks and valleys that somewhat mirror the metric we're comparing it to. (Of course, the exact time index for the product would vary on a week-by-week basis due to dynamic time warping.) Conversely, Product #716 (shown in the green stars) is the product with the worst match, showing almost no variability.

Finding the optimal product: Small DTW distance and similar absolute sales numbers

Now that we've developed a list of products that are closest to our factory's projected output (our "optimal sales trend"), we can filter them down to those that have small DTW distances as well as similar absolute sales numbers. One good candidate would be Product #202, which has a DTW distance of 6.86 versus the population median distance of 7.89 and tracks our optimal trend very closely.

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



Using MLflow to track best and worst products, along with artifacts

MLflow is an open source platform for managing the machine learning lifecycle, including experimentation, reproducibility and deployment. **Databricks notebooks offer a fully integrated MLflow environment, allowing you to create experiments, log parameters and metrics, and save results.** For more information about getting started with MLflow, take a look at the excellent documentation.

MLflow's design is centered around the ability to log all of the inputs and outputs of each experiment we do in a systematic, reproducible way. On every pass through the data, known as a "Run," we're able to log our experiment's:

- **Parameters:** The inputs to our model
- **Metrics:** The output of our model, or measures of our model's success
- **Artifacts:** Any files created by our model — for example, PNG plots or CSV data output
- **Models:** The model itself, which we can later reload and use to serve predictions

In our case, we can use it to run the dynamic time warping algorithm several times over our data while changing the "stretch factor," the maximum amount of warp that can be applied to our time series data. To initiate an MLflow experiment, and allow for easy logging using `mlflow.log_param()`, `mlflow.log_metric()`, `mlflow.log_artifact()`, and `mlflow.log_model()`, we wrap our main function using:

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

as shown in the abbreviated code at right.

```
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)
```

With each run through the data, we've created a log of the "stretch factor" parameter being used, and a log of products we classified as being outliers based upon the z-score of the DTW distance metric. We were even able to save an artifact (file) of a histogram of the DTW distances. **These experimental runs are saved locally on Databricks and remain accessible in the future if you decide to view the results of your experiment at a later date.**

Now that MLflow has saved the logs of each experiment, we can go back through and examine the results. From your Databricks notebook, select the "Runs" icon in the upper right-hand corner to view and compare the results of each of our runs.

Not surprisingly, as we increase our "stretch factor," our distance metric decreases. Intuitively, this makes sense: As we give the algorithm more flexibility to warp the time indices forward or backward, it will find a closer fit for the data. In essence, we've traded some bias for variance.

Logging models in MLflow

MLflow has the ability to not only log experiment parameters, metrics and artifacts (like plots or CSV files), but also to log machine learning models. An MLflow model is simply a folder that is structured to conform to a consistent API, ensuring compatibility with other MLflow tools and features. This interoperability is very powerful, allowing any Python model to be rapidly deployed to many different types of production environments.

MLflow comes preloaded with a number of common model "flavors" for many of the most popular machine learning libraries, including scikit-learn, Spark MLlib, PyTorch, TensorFlow and others. These model flavors make it trivial to log and reload models after they are initially constructed, as demonstrated in this blog post. For example, when using MLflow with scikit-learn, logging a model is as easy as running the following code from within an experiment:

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

MLflow also offers a "Python function" flavor, which allows you to save any model from a third-party library (such as XGBoost or spaCy), or even a simple Python function itself, as an MLflow model. Models created using the Python function flavor live within the same ecosystem and are able to interact with other MLflow tools through the Inference API. Although it's impossible to plan for every use case, the Python function model flavor was designed to be as universal and flexible as possible. It allows for custom processing and logic evaluation, which can come in handy for ETL applications. Even as more "official" model flavors come online, the generic Python function flavor will still serve as an important "catchall," providing a bridge between Python code of any kind and MLflow's robust tracking toolkit.

Logging a model using the Python function flavor is a straightforward process. **Any model or function can be saved as a model, with one requirement: It must take in a pandas DataFrame as input, and return a DataFrame or NumPy array.** Once that requirement is met, saving your function as an MLflow model involves defining a Python class that inherits from PythonModel, and overriding the `.predict() method` with your custom function, as described here.

Loading a logged model from one of our runs

Now that we've run through our data with several different stretch factors, the natural next step is to examine our results and look for a model that did particularly well according to the metrics that we've logged. **MLflow makes it easy to then reload a logged model and use it to make predictions on new data, using the following instructions:**

1. Click on the link for the run you'd like to load our model from
2. Copy the "Run ID"
3. Make note of the name of the folder the model is stored in. In our case, it's simply named "model"
4. Enter the model folder name and Run ID as shown below:

```
import custom_flavor as mlflow_custom_flavor

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

To show that our model is working as intended, we can now load the model and use it to measure DTW distances on two new products that we've created within the variable `new_sales_units`:

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

Next steps

As you can see, our MLflow model is predicting new and unseen values with ease. And since it conforms to the Inference API, we can deploy our model on any serving platform (such as Microsoft Azure ML or Amazon SageMaker), deploy it as a local REST API end point, or create a user-defined function (UDF) that can easily be used with Spark SQL. In closing, we demonstrated how we can use dynamic time warping to predict sales trends using the Databricks Unified Data Analytics Platform. Try out the Using Dynamic Time Warping and MLflow to Predict Sales Trends notebook with Databricks Runtime for Machine Learning today.

CHAPTER 4:

How a Fresh Approach to Safety Stock Analysis Can Optimize Inventory

By **Bryan Smith** and **Rob Saker**

April 22, 2020

Try this notebook in Databricks →

A manufacturer is working on an order for a customer only to find that the delivery of a critical part is delayed by a supplier. A retailer experiences a spike in demand for beer thanks to an unforeseen reason, and they lose sales because of their lack of supply. Customers have a negative experience because of your inability to meet demand. These companies lose immediate revenue and your reputation is damaged. Does this sound familiar?

In an ideal world, demand for goods would be easily predictable. In practice, even the best forecasts are impacted by unexpected events. Disruptions happen due to raw material supply, freight and logistics, manufacturing breakdowns, unexpected demand and more. Retailers, distributors, manufacturers and suppliers all must wrestle with these challenges to ensure they are able to reliably meet their customers' needs while also not carrying excessive inventory. This is where an improved method of safety stock analysis can help your business.

Organizations constantly work on allocating resources where they are needed to meet anticipated demand. The immediate focus is often in improving the accuracy of their forecasts. To achieve this goal, organizations are investing in scalable platforms, in-house expertise and sophisticated new models.

Even the best forecasts do not perfectly predict the future, and sudden shifts in demand can leave shelves bare. This was highlighted in early 2020 when concerns about the virus that causes COVID-19 led to widespread toilet paper stockouts. As Craig Boyan, the president of H-E-B commented, "We sold in two weeks what we normally sell in two months."

Scaling up production is not a simple solution to the problem. Georgia-Pacific, a leading manufacturer of toilet paper, estimated that the average American household would consume 40% more toilet paper as people stayed home during the pandemic. In response, the company was able to boost production by 20% across its 14 facilities configured for the production of toilet paper. Most mills already run operations 24 hours a day, seven days a week with fixed capacity, so any further increase in production would require an expansion in capacity enabled through the purchase of additional equipment or the building of new plants.

This bump in production output can have upstream consequences. Suppliers may struggle to provide the resources required by newly scaled and expanded manufacturing capacity. Toilet paper is a simple product, but its production depends on pulp shipped from forested regions of the U.S., Canada, Scandinavia and Russia as well as more locally sourced recycled paper fiber. It takes time for suppliers to harvest, process and ship the materials needed by manufacturers once initial reserves are exhausted.

A supply chain concept called the bullwhip effect underpins all this uncertainty. Distorted information throughout the supply chain can cause large inefficiencies in inventory, increased freight and logistics costs, inaccurate capacity planning and more. Manufacturers or retailers eager to return stocks to normal may trigger their suppliers to ramp production which in turn triggers upstream suppliers to ramp theirs. If not carefully managed, retailers and suppliers may find themselves with excess inventory and production capacity when demand returns to normal or even encounters a slight dip below normal as consumers work through a backlog of their own personal inventories. Careful consideration of the dynamics of demand along with scrutiny of the uncertainty around the demand we forecast is needed to mitigate this bullwhip effect.

Managing uncertainty with safety stock analysis

The kinds of shifts in consumer demand surrounding the COVID-19 pandemic are hard to predict, but they highlight an extreme example of the concept of uncertainty that every organization managing a supply chain must address. Even in periods of relatively normal consumer activity, demand for products and services varies and must be considered and actively managed against.

Modern demand forecasting tools predict a mean value for demand, taking into consideration the effects of weekly and annual seasonality, long-term trends, holidays and events, and external influencers such as weather, promotions, the economy and additional factors. They produce a singular value for forecasted demand that can be misleading, as half the time we expect to see demand below this value and the other half we expect to see demand above it.

The mean forecasted value is important to understand, but just as critical is an understanding of the uncertainty on either side of it. We can think of this uncertainty as providing a range of potential demand values, each of which has a quantifiable probability of being encountered. And by thinking of our forecasts this way, we can begin to have a conversation about what parts of this range we should attempt to address.

Statistically speaking, the full range of potential demand is infinite and, therefore, never 100% fully addressable. But long before we need to engage in any kind of theoretical dialogue, we can recognize that each incremental improvement in our ability to address the range of potential demand comes with a sizable (actually exponential) increase in inventory requirements. This leads us to pursue a targeted service level at which we attempt to address a specific proportion of the full range of possible demand that balances the revenue goals of our organization with the cost of inventory.

The consequence of defining this service level expectation is that we must carry a certain amount of extra inventory, above the volume required to address our mean forecasted demand, to serve as a buffer against uncertainty. This safety stock, when added to the cycle stock required to meet mean periodic demand, gives us the ability to address most (though not all) fluctuations in actual demand while balancing our overall organizational goals.

Calculating the required safety stock levels

In the classic supply chain literature, safety stock is calculated using one of two formulas that address uncertainty in demand and uncertainty in delivery. As our focus in this article is on demand uncertainty, we could eliminate the consideration of uncertain lead times, leaving us with a single, simplified safety stock formula to consider:

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

In a nutshell, this formula explains that safety stock is calculated as the average uncertainty in demand around the mean forecasted value (σ_D) multiplied by the square root of the duration of the (performance) cycle for which we are stocking ($\sqrt{PC/T}$) multiplied by a value associated with the portion of the range of uncertainty we wish to address (Z). Each component of this formula deserves a little explanation to ensure it is fully understood.

In the previous section of this article, we explained that demand exists as a range of potential values around a mean value, which is what our forecast generates. If we assume this range is evenly distributed around this mean, we can calculate an average of this range on either side of the mean value. This is known as a standard deviation. The value σ_D, also known as the standard deviation of demand, provides us with a measure of the range of values around the mean.

Because we have assumed this range is balanced around the mean, it turns out that we can derive the proportion of the values in this range that exist some number of standard deviations from that mean. If we use our service level expectation to represent the proportion of potential demand we wish to address, we can back into the number of standard deviations in demand that we need to consider as part of our planning for safety stock. The actual math behind the calculation of the required number of standard deviations (known as z-scores as represented in the formula as z) required to capture a percentage of the range of values gets a little complex, but luckily z-score tables are widely published and online calculators are available. With that said, here are some z-score values that correspond to some commonly employed service level expectations:

SERVICE LEVEL EXPECTATION	Z (Z-SCORE)
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

Finally, we get to the term that addresses the duration of the cycle for which we are calculating safety stock ($\sqrt{PC}/T$). Putting aside why it is we need the square root calculation, this is the simplest element of the formula to understand. The PC/T value represents the duration of the cycle for which we are calculating our safety stock. The division by T is simply a reminder that we need to express this duration in the same units as those used to calculate our standard deviation value. For example, if we were planning safety stock for a 7-day cycle, we can take the square root of 7 for this term so long as we have calculated the standard deviation of demand leveraging daily demand values.

Demand variance is hard to estimate

On the surface, the calculation of safety stock analysis requirements is fairly straightforward. In supply chain management classes, students are often provided historical values for demand from which they can calculate the standard deviation component of the formula. Given a service level expectation, they can then quickly derive a z-score and pull together the safety stock requirements to meet that target level. But these numbers are wrong, or at least they are wrong outside a critical assumption that is almost never valid.

The sticking point in any safety stock calculation is the standard deviation of demand. The standard formula depends on knowing the variation associated with demand in the future period for which we are planning. It is extremely rare that variation in a time series is stable. Instead, it often changes with trends and seasonal patterns in the data. Events and external regressors exert their own influences as well.

To overcome this problem, supply chain software packages often substitute measures of forecast error such as the root mean squared error (RMSE) or mean absolute error (MAE) for the standard deviation of demand, but these values represent different (though related) concepts. This often leads to an underestimation of safety stock requirements as is illustrated in this chart within which a 92.7% service level is achieved despite the setting of a 95% expectation.

And as most forecasting models work to minimize error while calculating a forecast mean, the irony is that improvements in model performance often exacerbate the problem of underestimation. It's very likely this is behind the growing recognition that although many retailers work toward published service level expectations, most of them fall short of these goals.

Where do we go from here and how does Databricks help?

An important first step in addressing the problem is recognizing the shortcomings in our safety stock analysis calculations. Recognition alone is seldom satisfying.

A few researchers are working to define techniques that better estimate demand variance for the explicit purpose of improving safety stock estimation, but there isn't consensus as to how this should be performed. And software to make these techniques easier to implement isn't widely available.

For now, we would strongly encourage supply chain managers to carefully examine their historical service level performance to see whether stated targets are being met. This requires the careful combination of past forecasts as well as historical actuals. Because of the cost of preserving data in traditional database platforms, many organizations do not keep past forecasts or atomic-level source data, but the use of cloud-based storage with data stored in high-performance, compressed formats accessed through on-demand computational technology — provided through platforms such as Databricks — can make this cost effective and provide improved query performance for many organizations.

As automated or digitally enabled fulfillment systems are deployed — required for many buy online pick up in-store (BOPIS) models — and begin generating real-time data on order fulfillment, companies will wish to use this data to detect out-of-stock issues that indicate the need to reassess service level expectations as well as in-store inventory management practices. Manufacturers that were limited to running these analyses on a daily routine may want to analyze and make adjustments per shift. Databricks streaming ingestion capabilities provide a solution, enabling companies to perform safety stock analysis with near real-time data.

Finally, consider exploring new methods of generating forecasts that provide better inputs into your inventory planning processes. The combination of using Facebook Prophet with parallelization and autoscaling platforms such as Databricks has allowed organizations to make timely, fine-grain forecasting a reality for many enterprises. Still other forecasting techniques, such as Generalized Autoregressive Conditional Heteroskedastic (GARCH) models, may allow you to examine shifts in demand variability that could prove very fruitful in designing a safety stock strategy.

The resolution of the safety stock challenge has significant potential benefits for organizations willing to undertake the journey, but as the path to the end state is not readily defined, flexibility is going to be the key to your success. We believe that Databricks is uniquely positioned to be the vehicle for this journey, and we look forward to working with our customers in navigating it together.

Databricks thanks Professor Sreekumar Bhaskaran at the Southern Methodist University Cox School of Business for his insights on this important topic.

> Start experimenting with this free Databricks notebook.

CHAPTER 5:

New Methods for Improving Supply Chain Demand Forecasting

Fine-grain demand forecasting with causal factors

By **Bryan Smith** and **Rob Saker**

March 26, 2020

Organizations are rapidly embracing fine-grain demand forecasting

Retailers and consumer goods manufacturers are increasingly seeking improvements to their supply chain management in order to reduce costs, free up working capital and create a foundation for omnichannel innovation. Changes in consumer purchasing behavior are placing new strains on the supply chain. Developing a better understanding of consumer demand via a demand forecast is considered a good starting point for most of these efforts as the demand for products and services drives decisions about the labor, inventory management, supply and production planning, freight and logistics and many other areas.

In *Notes from the AI Frontier*, McKinsey & Company highlight that a 10% to 20% improvement in retail supply chain forecasting accuracy is likely to produce a 5% reduction in inventory costs and a 2% to 3% increase in revenues. Traditional supply chain forecasting tools have failed to deliver the desired results. With claims of industry-average inaccuracies of 32% in retailer supply chain demand forecasting, the potential impact of even modest forecasting improvements is immense for most retailers. As a result, many organizations are moving away from prepackaged forecasting solutions, exploring ways to bring demand forecasting skills in-house and revisiting past practices that compromised forecast accuracy for computational efficiency.

A key focus of these efforts is the generation of forecasts at a finer level of temporal and (location/product) hierarchical granularity. Fine-grain demand forecasts have the potential to capture the patterns that influence demand closer to the level at which that demand must be met. Whereas in the past, a retailer might have predicted short-term demand for a class of products at a market level or distribution level, for a month or week period, and then used the forecasted values to allocate how units of a specific product in that class should be placed in a given store and day, fine-grain demand forecasting allows forecasters to build more localized models that reflect the dynamics of that specific product in a particular location.

Fine-grain demand forecasting comes with challenges

As exciting as fine-grain demand forecasting sounds, it comes with many challenges. First, by moving away from aggregate forecasts, the number of forecasting models and predictions that must be generated explodes. The level of processing required is either unattainable by existing forecasting tools, or it greatly exceeds the service windows for this information to be useful. This limitation leads to companies making trade-offs in the number of categories being processed or the level of grain in the analysis.

As examined in a prior blog post, Apache Spark™ can be employed to overcome this challenge, allowing modelers to parallelize the work for timely, efficient execution. When deployed on cloud-native platforms such as Databricks, computational resources can be quickly allocated and then released, keeping the cost of this work within budget.

The second and more difficult challenge to overcome is understanding that demand patterns that exist in aggregate may not be present when examining data at a finer level of granularity. To paraphrase Aristotle, the whole may often be greater than the sum of its parts. As we move to lower levels of detail in our analysis, patterns more easily modeled at higher levels of granularity may no longer be reliably present, making the generation of forecasts with techniques applicable at higher levels more challenging. This problem within the context of forecasting is noted by many practitioners going all the way back to Henri Theil in the 1950s.

As we move closer to the transaction level of granularity, we also need to consider the external causal factors that influence individual customer demand and purchase decisions. In aggregate, these may be reflected in the averages, trends and seasonality that make up a time series, but at finer levels of granularity, we may need to incorporate these directly into our forecasting models.

Finally, moving to a finer level of granularity increases the likelihood the structure of our data will not allow for the use of traditional forecasting techniques. The closer we move to the transaction grain, the higher the likelihood we will need to address periods of inactivity in our data. At this level of granularity, our dependent variables, especially when dealing with count data such as units sold, may take on a skewed distribution that's not amenable to simple transformations and which may require the use of forecasting techniques outside the comfort zone of many data scientists.

Accessing the historical data

See the Data Preparation notebook for details →

In order to examine these challenges, we will leverage public trip history data from the New York City Bike Share program, also known as Citi Bike NYC. Citi Bike NYC is a company that promises to help people "Unlock a Bike. Unlock New York." Their service allows people to go to any of over 850 various rental locations throughout the NYC area and rent bikes. The company has an inventory of over 13,000 bikes with plans to increase the number to 40,000. Citi Bike has well over 100,000 subscribers who make nearly 14,000 rides per day.

Citi Bike NYC reallocates bikes from where they were left to where they anticipate future demand. Citi Bike NYC has a challenge that is similar to what retailers and consumer goods companies deal with on a daily basis. How do we best predict demand to allocate resources to the right areas? If we underestimate demand, we miss revenue opportunities and potentially hurt customer sentiment. If we overestimate demand, we have excess bike inventory being unused.

This publicly available data set provides information on each bicycle rental from the end of the prior month all the way back to the inception of the program in mid-2013. The trip history data identifies the exact time a bicycle is rented from a specific rental station and the time that bicycle is returned to another rental station. If we treat stations in the Citi Bike NYC program as store locations and consider the initiation of a rental as a transaction, we have something closely approximating a long and detailed transaction history with which we can produce forecasts.

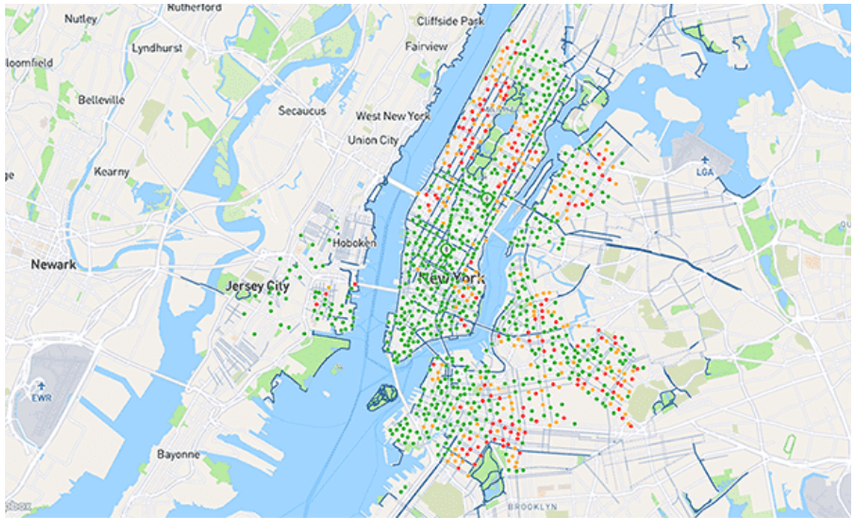
As part of this exercise, we will need to identify external factors to incorporate into our modeling efforts. We will leverage both holiday events as well as historical (and predicted) weather data as external influencers. For the holiday data set, we will simply identify standard holidays from 2013 to present using the holidays library in Python. For the weather data, we will employ hourly extracts from Visual Crossing, a popular weather data aggregator.

Citi Bike NYC and Visual Crossing data sets have terms and conditions that prohibit our directly sharing their data. Those wishing to recreate our results should visit the data providers' websites, review their Terms & Conditions, and download their data sets to their environments in an appropriate manner. We will provide the data preparation logic required to transform these raw data assets into the data objects used in our analysis.

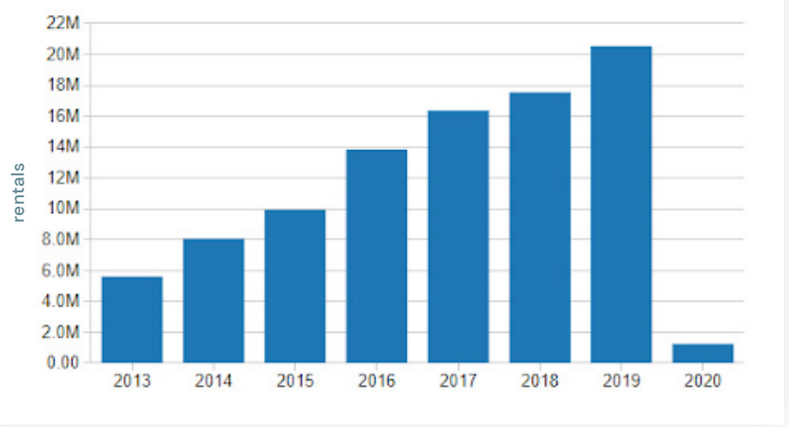
Examining the transactional data

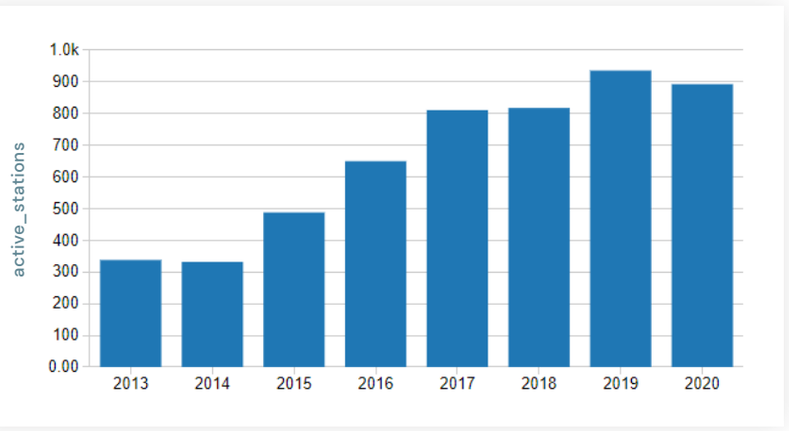
See the Exploratory Analysis notebook for details →

As of January 2020, the Citi Bike NYC bike share program consisted of 864 active stations operating in the New York City metropolitan area, primarily in Manhattan. In 2019 alone, a little over 4 million unique rentals were initiated by customers with as many as nearly 14,000 rentals taking place on peak days.

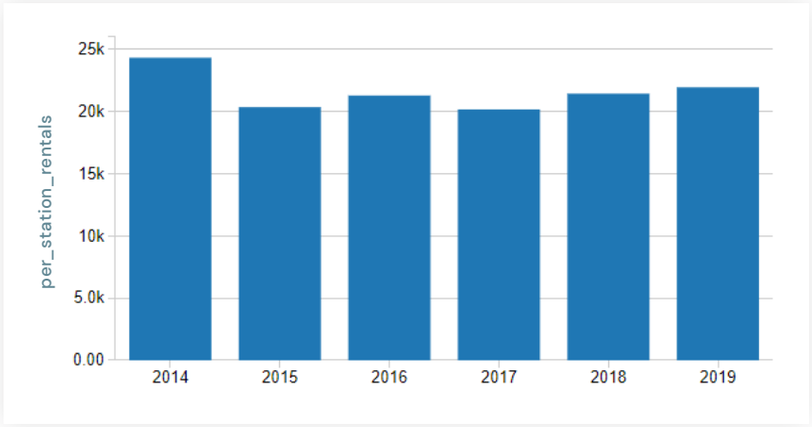


Since the start of the program, we can see the number of rentals has increased year over year. Some of this growth is likely due to the increased utilization of the bicycles, but much of it seems to be aligned with the expansion of the overall station network.

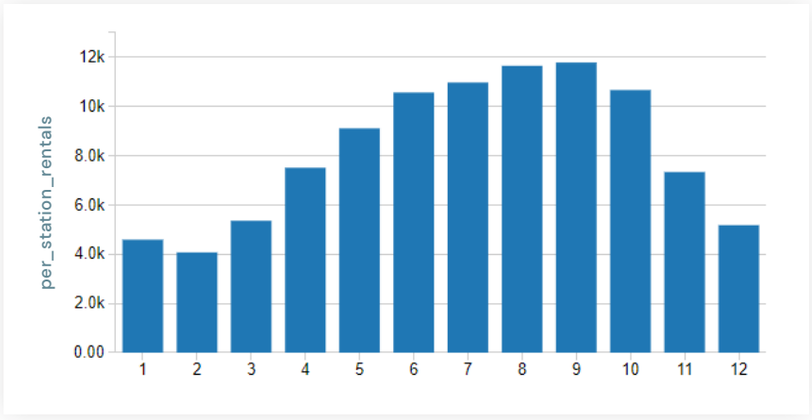




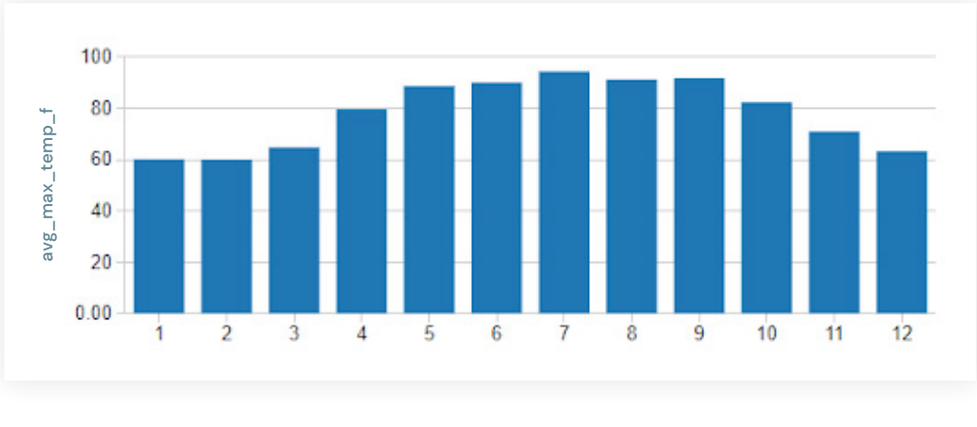
Normalizing rentals by the number of active stations in the network shows that growth in ridership on a per-station basis has been slowly ticking up for the last few years in what we might consider to be a slight linear upward trend.



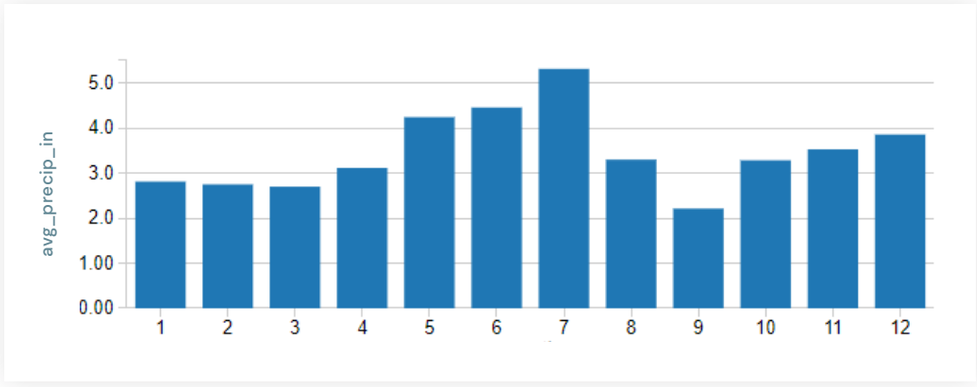
Using this normalized value for rentals, ridership seems to follow a distinctly seasonal pattern, rising in the spring, summer and fall and then dropping in winter as the weather outside becomes less conducive to bike riding.



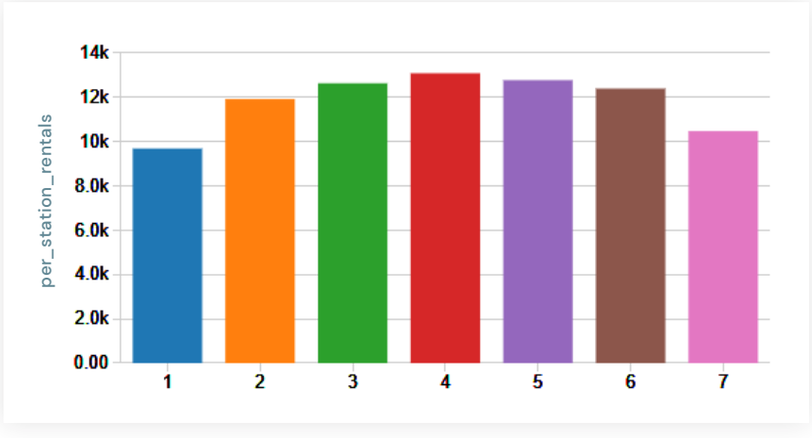
This pattern appears to closely follow patterns in the maximum temperatures (in degrees Fahrenheit) for the city.



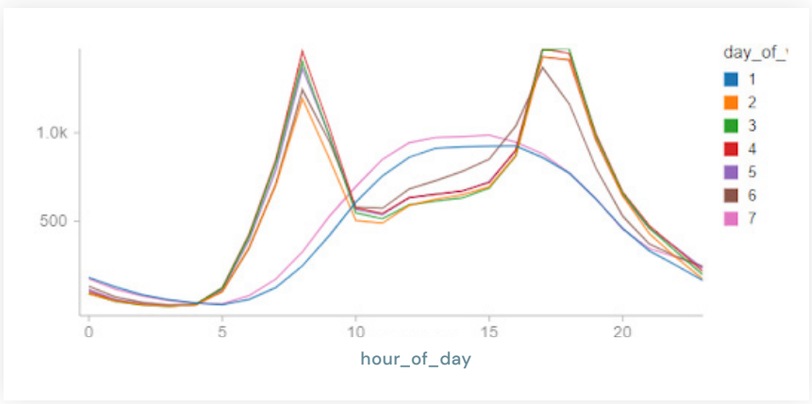
While it can be hard to separate monthly ridership from patterns in temperatures, rainfall (in average monthly inches) does not mirror these patterns quite so readily.



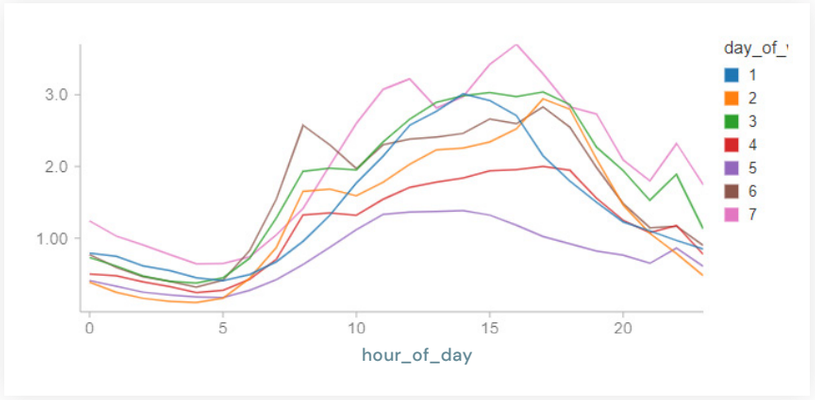
Examining weekly patterns of ridership with Sunday identified as 1 and Saturday identified as 7, it would appear that New Yorkers are using the bicycles as commuter devices, a pattern seen in many other bike share programs.



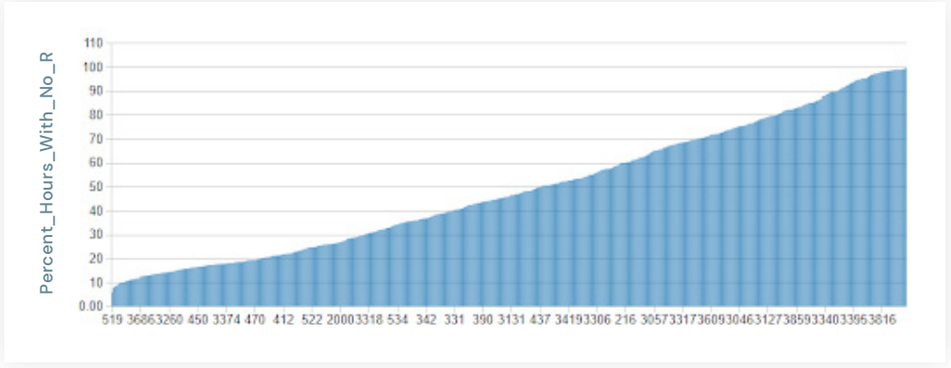
Breaking down these ridership patterns by hour of the day, we see distinct weekday patterns where ridership spikes during standard commute hours. On the weekends, patterns indicate more leisurely utilization of the program, supporting our earlier hypothesis.



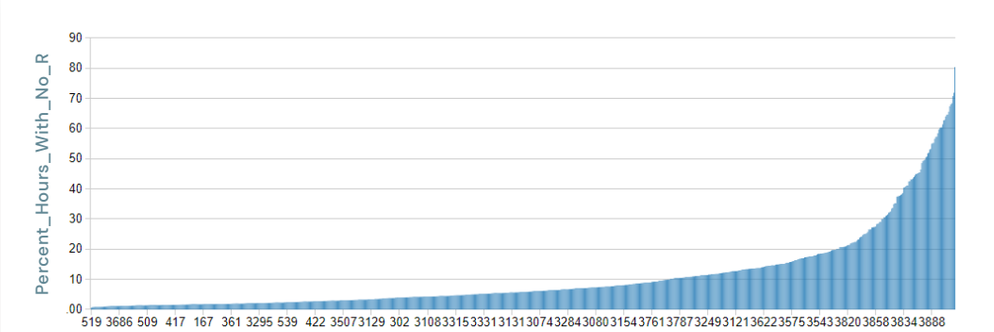
An interesting pattern is that holidays, regardless of their day of week, show consumption patterns that roughly mimic weekend usage patterns. The infrequent occurrence of holidays may be the cause of erraticism of these trends. Still, the chart seems to support that the identification of holidays is important to producing a reliable forecast.



In aggregate, the hourly data appear to show that New York City is truly the city that never sleeps. In reality, there are many stations for which there are a large proportion of hours during which no bicycles are rented.



These gaps in activity can be problematic when attempting to generate a forecast. By moving from 1-hour to 4-hour intervals, the number of periods within which individual stations experience no rental activity drops considerably, though there are still many stations that are inactive across this time frame.



Instead of ducking the problem of inactive periods by moving toward even higher levels of granularity, we will attempt to make a forecast at the hourly level, exploring how an alternative forecasting technique may help us deal with this data set. As forecasting for stations that are largely inactive isn't terribly interesting, we'll limit our analysis to the top 200 most active stations.

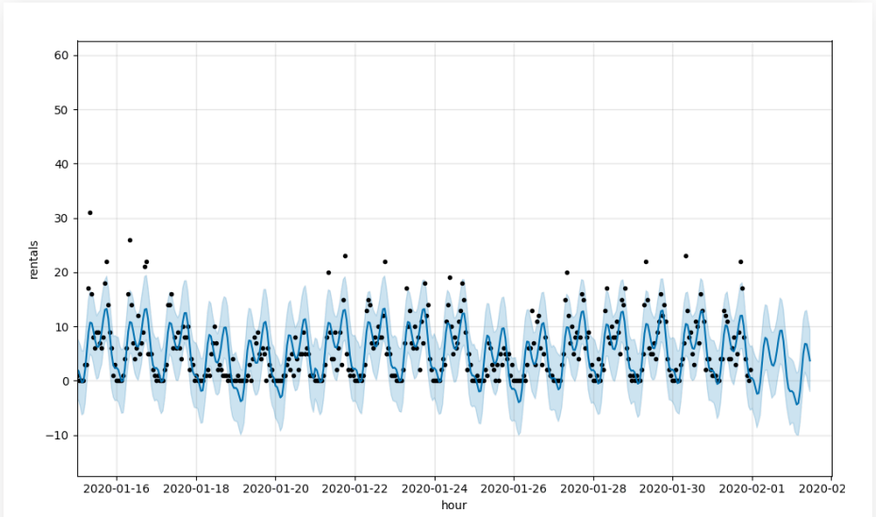
Forecasting bike share rentals with Facebook Prophet

In an initial attempt to forecast bike rentals at the per-station level, we made use of Facebook Prophet, a popular Python library for time series forecasting. The model was configured to explore a linear growth pattern with daily, weekly and yearly seasonal patterns. Periods in the data set associated with holidays were also identified so that anomalous behavior on these dates would not affect the average, trend and seasonal patterns detected by the algorithm.

Using the scale-out pattern documented in the previously referenced blog post, models were trained for the most active 200 stations, and 36-hour forecasts were generated for each. Collectively, the models had a Root Mean Squared Error (RMSE) of 5.44 with a Mean Average Proportional Error (MAPE) of 0.73. (Zero-value actuals were adjusted to 1 for the MAPE calculation.)

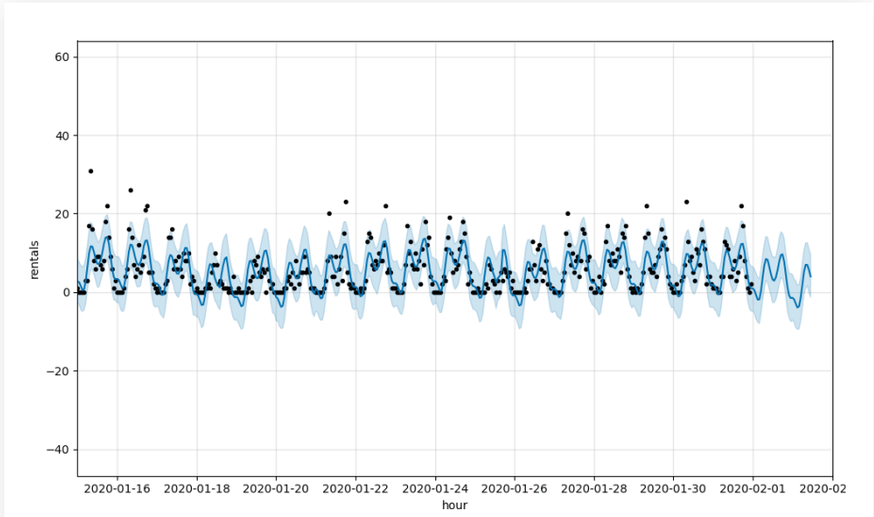
These metrics indicate that the models do a reasonably good job of predicting rentals but are missing when hourly rental rates move higher. Visualizing sales data for individual stations, you can see this graphically such as in this chart for Station 518, E 39 St and 2 Ave, which has a RMSE of 4.58 and a MAPE of 0.69:

See the Time Series notebook for details →



The model was then adjusted to incorporate temperature and precipitation as regressors. Collectively, the resulting forecasts had a RMSE of 5.35 and a MAPE of 0.72. While a very slight improvement, the models are still having difficulty picking up on the large swings in ridership found at the station level, as demonstrated again by Station 518, which had a RMSE of 4.51 and a MAPE of 0.68:

See the Time Series With Regressors notebook for details →



This pattern of difficulty modeling the higher values in both the time series models is typical of working with data having a Poisson distribution. In such a distribution, we will have a large number of values around an average with a long tail of values above it. On the other side of the average, a floor of zero leaves the data skewed. Today, Facebook Prophet expects data to have a normal (Gaussian) distribution but plans for the incorporation of Poisson regression have been discussed.

Alternative approaches to forecasting supply chain demand

How might we then proceed with generating a forecast for these data? One solution, as the caretakers of Facebook Prophet are considering, is to leverage Poisson regression capabilities in the context of a traditional time series model. While this may be an excellent approach, it is not widely documented, so tackling this on our own before considering other techniques may not be the best approach for our needs.

Another potential solution is to model the scale of non-zero values and the frequency of the occurrence of the zero-valued periods. The output of each model can then be combined to assemble a forecast. This method, known as Croston's method, is supported by the recently released Croston Python library while another data scientist has implemented his own function for it. Still, this is not a widely adopted method (despite the technique dating back to the 1970s) and our preference is to explore something a bit more out of the box.

Given this preference, a random forest regressor would seem to make quite a bit of sense. Decision trees, in general, do not impose the same constraints on data distribution as many statistical methods. The range of values for the predicted variable is such that it may make sense to transform rentals using something like a square root transformation before training the model, but even then, we might see how well the algorithm performs without it.

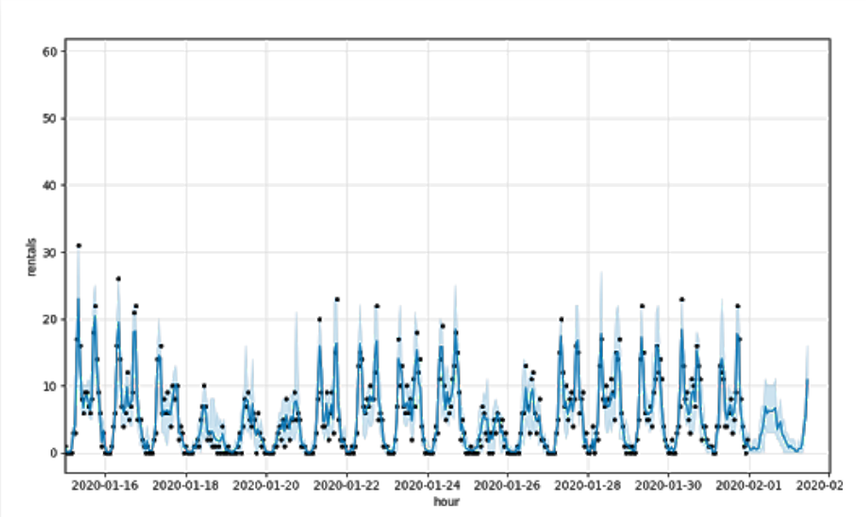
To leverage this model, we'll need to engineer a few features. It's clear from the exploratory analysis that there are strong seasonal patterns in the data, both at the annual, weekly and daily levels. This leads us to extract year, month, day of week and hour of the day as features. We may also include a flag for holiday.

Using a random forest regressor and nothing but time-derived features, we arrive at an overall RMSE of 3.4 and MAPE of 0.39. For Station 518, the RMSE and MAPE values are 3.09 and 0.38, respectively:

See the Temporal notebook for details →

By leveraging precipitation and temperature data in combination with some of these same temporal features, we are able to better (though not perfectly) address some of the higher rental values. The RMSE for Station 518 drops to 2.14 and the MAPE to 0.26. Overall, the RMSE drops to 2.37 and MAPE to 0.26 indicating weather data is valuable in forecasting demand for bicycles.

See the Random Forest With Temporal & Weather Features notebook for details →



Implications of the results

Demand forecasting at finer levels of granularity may require us to think differently about our approach to modeling. External influencers, which may be safely considered summarized in high-level time series patterns, may need to be more explicitly incorporated into our models. Patterns in data distribution hidden at the aggregate level may become more readily exposed and necessitate changes in modeling approaches. In this data set, these challenges were best addressed by the inclusion of hourly weather data and a shift away from traditional time series techniques toward an algorithm that makes fewer assumptions about our input data.

There may be many other external influencers and algorithms worth exploring, and as we go down this path, we may find that some of these work better for some subset of our data than for others. We may also find that as new data arrives, techniques that previously worked well may need to be abandoned and new techniques considered.

A common pattern we are seeing with customers exploring fine-grain demand forecasting is the evaluation of multiple techniques with each training and forecasting cycle, something we might describe as an automated model bake-off. In a bake-off round, the model producing the best results for a given subset of the data wins the round with each subset able to decide its own winning model type. In the end, we want to ensure we are performing good data science where our data is properly aligned with the algorithms we employ, but as is noted in article after article, there isn't always just one solution to a problem and some may work better at one time than at others. The power of what we have available today with platforms like Apache Spark and Databricks is that we have access to the computational capacity to explore all these paths and deliver the best solution to our business.

Additional Retail/CPG and demand forecasting resources

Start experimenting with these developer resources:

1. Notebooks:
   - Data Preparation notebook
   - Exploratory Analysis notebook
   - Time Series notebook
   - Time Series With Regressors notebook
   - Temporal notebook
   - Random Forest With Temporal & Weather Features notebook
2. Download our Guide to Data Analytics and AI at Scale for Retail and CPG
3. Visit our Retail and CPG page to learn how Dollar Shave Club and Zalando are innovating with Databricks
4. Read our recent blog Fine-Grain Time Series Forecasting at Scale With Facebook Prophet and Apache Spark to learn how the Databricks Unified Data Analytics Platform addresses challenges in a timely manner and at a level of granularity that allows the business to make precise adjustments to product inventories

CHAPTER 6:

Fine-Grain Time Series Forecasting at Scale With Prophet and Apache Spark™

By **Bilal Obeidat**, **Bryan Smith** and **Brenner Heintz**

January 27, 2020

Try this time series forecasting notebook in Databricks →

Advances in time series forecasting are enabling retailers to generate more reliable demand forecasts. The challenge now is to produce these forecasts in a timely manner and at a level of granularity that allows the business to make precise adjustments to product inventories. Leveraging Apache Spark and Facebook Prophet, more and more enterprises facing these challenges are finding they can overcome the scalability and accuracy limits of past solutions.

In this post, we'll discuss the importance of time series forecasting, visualize some sample time series data, then build a simple model to show the use of Facebook Prophet. Once you're comfortable building a single model, we'll combine Prophet with the magic of Apache Spark to show you how to train hundreds of models at once, allowing us to create precise forecasts for each individual product-store combination at a level of granularity rarely achieved until now.

Accurate and timely forecasting is now more important than ever

Improving the speed and accuracy of time series analyses in order to better forecast demand for products and services is critical to retailers' success. If too much product is placed in a store, shelf and storeroom space can be strained, products can expire, and retailers may find their financial resources are tied up in inventory, leaving them unable to take advantage of new opportunities generated by manufacturers or shifts in consumer patterns. If too little product is placed in a store, customers may not be able to purchase the products they need. Not only do these forecast errors result in an immediate loss of revenue to the retailer, but over time consumer frustration may drive customers toward competitors.

New expectations require more precise time series forecasting methods and models

For some time, enterprise resource planning (ERP) systems and third-party solutions have provided retailers with demand forecasting capabilities based upon simple time series models. But with advances in technology and increased pressure in the sector, many retailers are looking to move beyond the linear models and more traditional algorithms historically available to them.

PROPHET

New capabilities, such as those provided by Facebook Prophet, are emerging from the data science community, and companies are seeking the flexibility to apply these machine learning models to their time series forecasting needs.

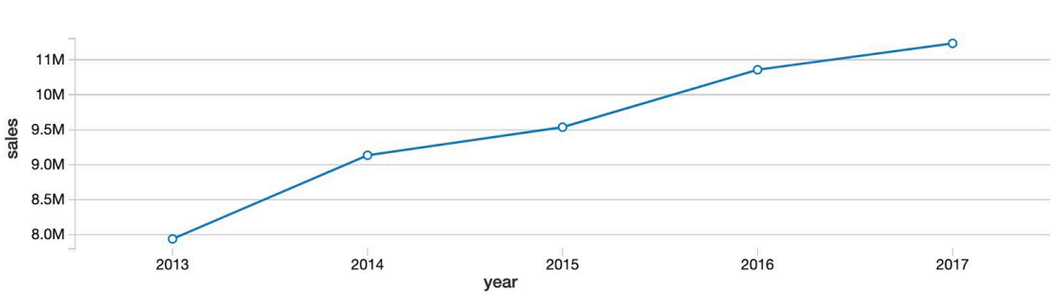
databricks

This movement away from traditional forecasting solutions requires retailers and the like to develop in-house expertise not only in the complexities of demand forecasting but also in the efficient distribution of the work required to generate hundreds of thousands or even millions of machine learning models in a timely manner. Luckily, we can use Spark to distribute the training of these models, making it possible to predict not just overall demand for products and services, but the unique demand for each product in each location.

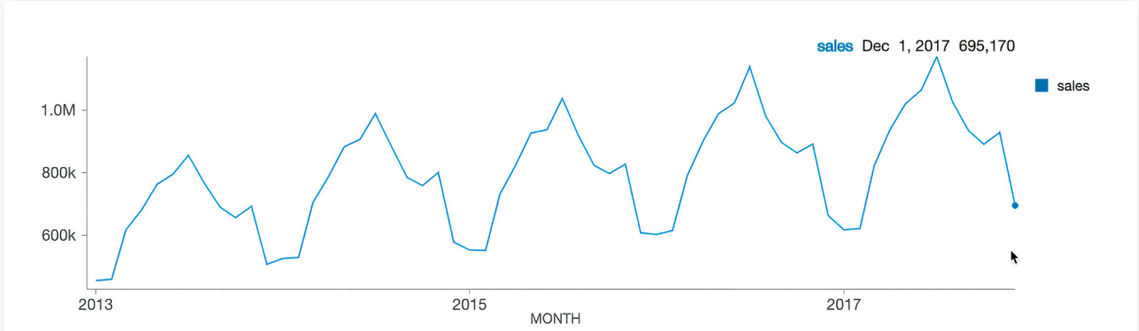
Visualizing demand seasonality in time series data

To demonstrate the use of Prophet to generate fine-grain demand forecasts for individual stores and products, we will use a publicly available data set from Kaggle. It consists of 5 years of daily sales data for 50 individual items across 10 different stores.

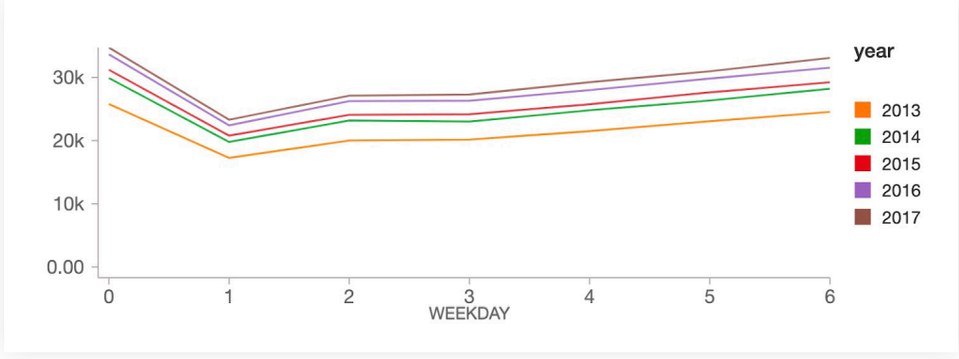
To get started, let's look at the overall yearly sales trend for all products and stores. As you can see, total product sales are increasing year over year with no clear sign of convergence around a plateau.



Next, by viewing the same data on a monthly basis, we can see that the year-over-year upward trend doesn't progress steadily each month. Instead, we see a clear seasonal pattern of peaks in the summer months and troughs in the winter months. Using the built-in data visualization feature of Databricks Collaborative Notebooks, we can see the value of our data during each month by mousing over the chart.



At the weekday level, sales peak on Sundays (weekday 0), followed by a hard drop on Mondays (weekday 1), then steadily recover throughout the rest of the week.



Getting started with a simple time series forecasting model on Facebook Prophet

As illustrated in the charts above, our data shows a clear year-over-year upward trend in sales, along with both annual and weekly seasonal patterns. It's these overlapping patterns in the data that Prophet is designed to address.

Facebook Prophet follows the scikit-learn API, so it should be easy to pick up for anyone with experience with sklearn. We need to pass in a two-column pandas DataFrame as input: the first column is the date, and the second is the value to predict (in our case, sales). Once our data is in the proper format, building a model is easy:

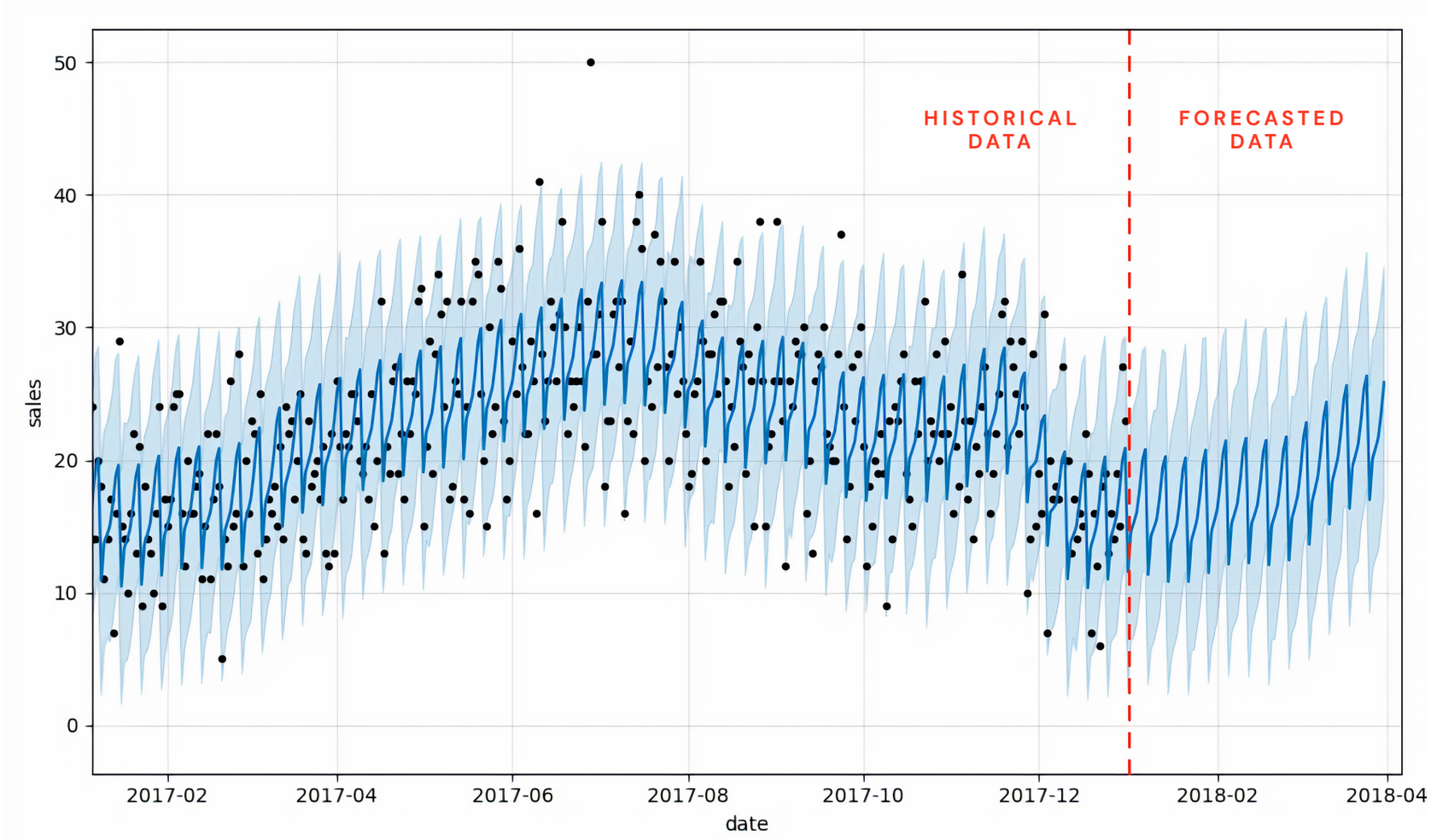
```
import pandas as pd
from fbprophet import Prophet

# 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)
```

Now that we have fit our model to the data, let's use it to build a 90-day forecast. In the code below, we define a data set that includes both historical dates and 90 days beyond, using Prophet's `make_future_dataframe` method:

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

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

That's it! We can now visualize how our actual and predicted data line up as well as a forecast for the future using Prophet's built-in .plot method. As you can see, the weekly and seasonal demand patterns we illustrated earlier are in fact reflected in the forecasted results.

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



This visualization is a bit busy. Bartosz Mikulski provides an excellent breakdown of it that is well worth checking out. In a nutshell, the black dots represent our actuals with the darker blue line representing our predictions and the lighter blue band representing our (95%) uncertainty interval.

Training hundreds of time series forecasting models in parallel with Prophet and Spark

Now that we've demonstrated how to build a single time series forecasting model, we can use the power of Apache Spark to multiply our efforts. Our goal is to generate not one forecast for the entire data set, but hundreds of models and forecasts for each product-store combination, something that would be incredibly time consuming to perform as a sequential operation.

Building models in this way could allow a grocery store chain, for example, to create a precise forecast for the amount of milk they should order for their Sandusky store that differs from the amount needed in their Cleveland store, based upon the differing demand at those locations.

How to use Spark DataFrames to distribute the processing of time series data

Data scientists frequently tackle the challenge of training large numbers of models using a distributed data processing engine such as Apache Spark. By leveraging a Spark cluster, individual worker nodes in the cluster can train a subset of models in parallel with other worker nodes, greatly reducing the overall time required to train the entire collection of time series models.

Of course, training models on a cluster of worker nodes (computers) requires more cloud infrastructure, and this comes at a price. But with the easy availability of on-demand cloud resources, companies can quickly provision the resources they need, train their models and release those resources just as quickly, allowing them to achieve massive scalability without long-term commitments to physical assets.

The key mechanism for achieving distributed data processing in Spark is the DataFrame. By loading the data into a Spark DataFrame, the data is distributed across the workers in the cluster. This allows these workers to process subsets of the data in a parallel manner, reducing the overall amount of time required to perform our work.

Of course, each worker needs to have access to the subset of data it requires to do its work. By grouping the data on key values, in this case on combinations of store and item, we bring together all the time series data for those key values onto a specific worker node.

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

> We share the groupBy code here to underscore how it enables us to train many models in parallel efficiently, although it will not actually come into play until we set up and apply a UDF to our data in the next section.

Leveraging the power of pandas user-defined functions

With our time series data properly grouped by store and item, we now need to train a single model for each group. To accomplish this, we can use a pandas user-defined function (UDF), which allows us to apply a custom function to each group of data in our DataFrame.

This UDF will not only train a model for each group, but also generate a result set representing the predictions from that model. But while the function will train and predict on each group in the DataFrame independent of the others, the results returned from each group will be conveniently collected into a single resulting DataFrame. This will allow us to generate store-item level forecasts but present our results to analysts and managers as a single output data set.

As you can see in the following abbreviated Python code, building our UDF is relatively straightforward. The UDF is instantiated with the `pandas_udf` method that identifies the schema of the data it will return and the type of data it expects to receive. Immediately following this, we define the function that will perform the work of the UDF.

Within the function definition, we instantiate our model, configure it and fit it to the data it has received. The model makes a prediction, and that data is returned as the output of the function.

```
@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
```

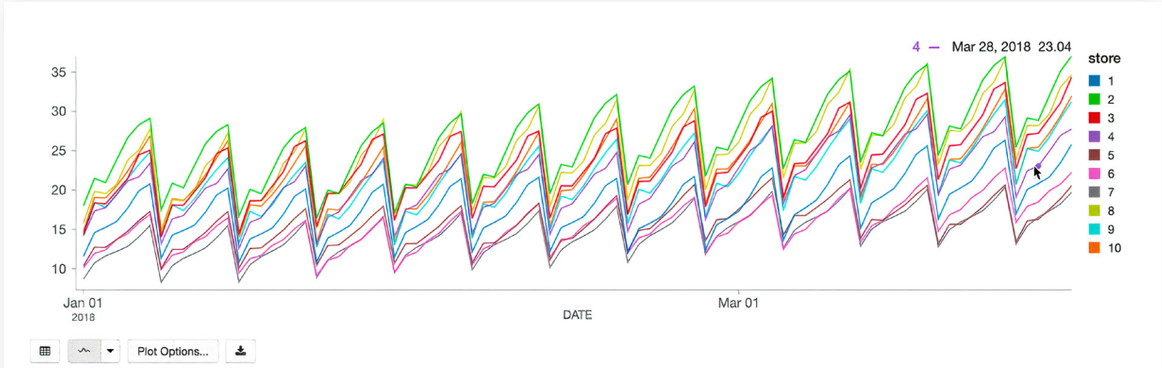
Now, to bring it all together, we use the `groupBy` command we discussed earlier to ensure our data set is properly partitioned into groups representing specific store and item combinations. We then simply `apply` the UDF to our DataFrame, allowing the UDF to fit a model and make predictions on each grouping of data.

The data set returned by the application of the function to each group is updated to reflect the date on which we generated our predictions. This will help us keep track of data generated during different model runs as we eventually take our functionality into production.

```
from pyspark.sql.functions import current_date

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

Next steps

We have now constructed a time series forecasting model for each store-item combination. Using a SQL query, analysts can view the tailored forecasts for each product. In the chart below, we've plotted the projected demand for product #1 across 10 stores. As you can see, the demand forecasts vary from store to store, but the general pattern is consistent across all of the stores, as we would expect.



As new sales data arrives, we can efficiently generate new forecasts and append these to our existing table structures, allowing analysts to update the business's expectations as conditions evolve.

To learn more, watch the on-demand webinar entitled How Starbucks Forecasts Demand at Scale With Facebook Prophet and Azure Databricks.

CHAPTER 7:

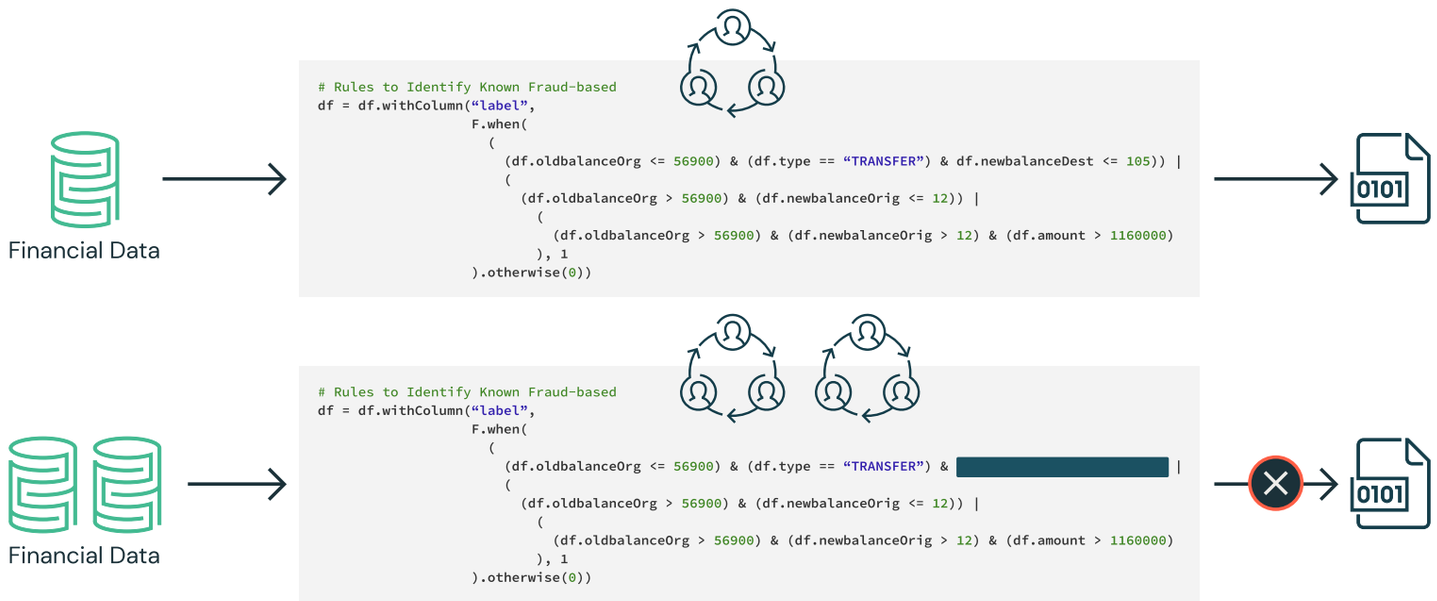
Detecting Financial Fraud at Scale With Decision Trees and MLflow on Databricks

By **Elena Boiarskaia**, **Navin Albert** and **Denny Lee**

May 2, 2019

Try this notebook in Databricks →

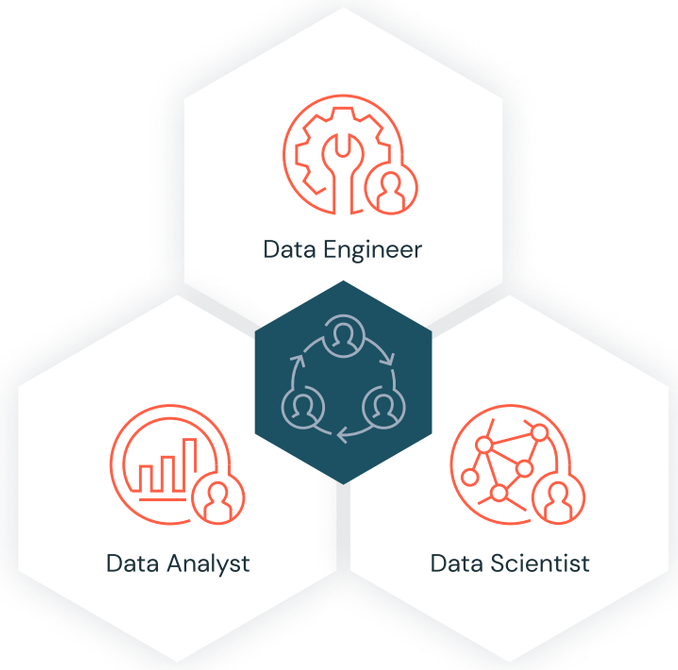
Detecting fraudulent patterns at scale using artificial intelligence is a challenge, no matter the use case. The massive amounts of historical data to sift through, the complexity of the constantly evolving machine learning and deep learning techniques, and the very small number of actual examples of fraudulent behavior are comparable to finding a needle in a haystack while not knowing what the needle looks like. In the financial services industry, the added concerns with security and the importance of explaining how fraudulent behavior was identified further increase the complexity of the task.



To build these detection patterns, a team of domain experts comes up with a set of rules based on how fraudsters typically behave. A workflow may include a subject matter expert in the financial fraud detection space putting together a set of requirements for a particular behavior. A data scientist may then take a subsample of the available data and select a set of deep learning or machine learning algorithms using these requirements and possibly some known fraud cases. To put the pattern in production, a data engineer may convert the resulting model to a set of rules with thresholds, often implemented using SQL.

databricks

This approach allows the financial institution to present a clear set of characteristics that lead to the identification of a fraudulent transaction that is compliant with the General Data Protection Regulation (GDPR). However, this approach also poses numerous difficulties. The implementation of a fraud detection system using a hardcoded set of rules is very brittle. Any changes to the fraud patterns would take a very long time to update. This, in turn, makes it difficult to keep up with and adapt to the shift in fraudulent activities that are happening in the current marketplace.

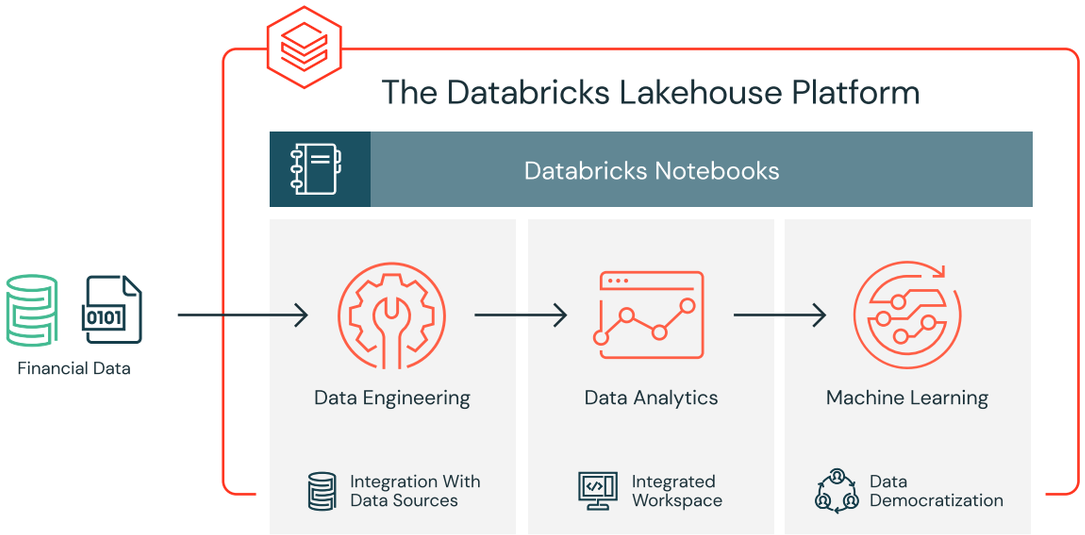


Additionally, the systems in the workflow described above are often siloed, with the domain experts, data scientists and data engineers all compartmentalized. The data engineer is responsible for maintaining massive amounts of data and translating the work of the domain experts and data scientists into production level code. Due to a lack of a common platform, the domain experts and data scientists have to rely on sampled down data that fits on a single machine for analysis. This leads to difficulty in communication and ultimately a lack of collaboration.

In this blog, we will showcase how to convert several such rule-based detection use cases to machine learning use cases on the Databricks platform, unifying the key players in fraud detection: domain experts, data scientists and data engineers. We will learn how to create a machine learning fraud detection data pipeline and visualize the data in real time, leveraging a framework for building modular features from large data sets. We will also learn how to detect fraud using decision trees and Apache Spark™ MLlib. We will then use MLflow to iterate and refine the model to improve its accuracy.

Solving with machine learning

There is a certain degree of reluctance with regard to machine learning models in the financial world as they are believed to offer a "black box" solution with no way of justifying the identified fraudulent cases. GDPR requirements, as well as financial regulations, make it seemingly impossible to leverage the power of data science. However, several successful use cases have shown that applying machine learning to detect fraud at scale can solve a host of the issues mentioned above.



Training a supervised machine learning model to detect financial fraud is very difficult due to the low number of actual confirmed examples of fraudulent behavior. However, the presence of a known set of rules that identify a particular type of fraud can help create a set of synthetic labels and an initial set of features. The output of the detection pattern that has been developed by the domain experts in the field has likely gone through the appropriate approval process to be put in production. It produces the expected fraudulent behavior flags and may, therefore, be used as a starting point to train a machine learning model. This simultaneously mitigates three concerns:

1. The lack of training labels
2. The decision of what features to use
3. Having an appropriate benchmark for the model

Training a machine learning model to recognize the rule-based fraudulent behavior flags offers a direct comparison with the expected output via a confusion matrix. Provided that the results closely match the rule-based detection pattern, this approach helps gain confidence in machine learning–based fraud prevention with the skeptics. The output of this model is very easy to interpret and may serve as a baseline discussion of the expected false negatives and false positives when compared to the original detection pattern.

Furthermore, the concern with machine learning models being difficult to interpret may be further assuaged if a decision tree model is used as the initial machine learning model. Because the model is being trained to a set of rules, the decision tree is likely to outperform any other machine learning model. The additional benefit is, of course, the utmost transparency of the model, which will essentially show the decision-making process for fraud, but without human intervention and the need to hard code any rules or thresholds. Of course, it must be understood that the future iterations of the model may utilize a different algorithm altogether to achieve maximum accuracy. The transparency of the model is ultimately achieved by understanding the features that went into the algorithm. Having interpretable features will yield interpretable and defensible model results.

The biggest benefit of the machine learning approach is that after the initial modeling effort, future iterations are modular and updating the set of labels, features or model type is very easy and seamless, reducing the time to production. This is further facilitated on the Databricks Collaborative Notebooks where the domain experts, data scientists and data engineers may work off the same data set at scale and collaborate directly in the notebook environment. So let's get started!

Ingesting and exploring the data

We will use a synthetic data set for this example. To load the data set yourself, please download it to your local machine from Kaggle and then import the data via Import Data — Azure and AWS.

The PaySim data simulates mobile money transactions based on a sample of real transactions extracted from one month of financial logs from a mobile money service implemented in an African country. The below table shows the information that the data set provides:

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).

Exploring the data

Creating the DataFrames: Now that we have uploaded the data to Databricks File System (DBFS), we can quickly and easily create DataFrames using Spark SQL.

```
# 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")
```

Now that we have created the DataFrame, let's take a look at the schema and the first thousand rows to review the data.

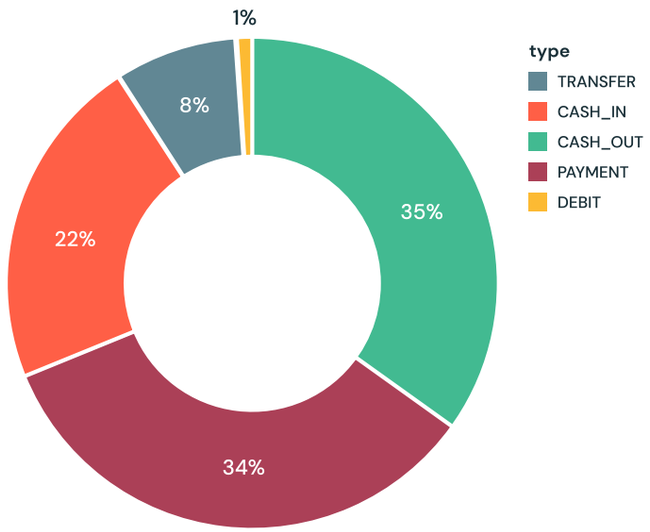
```
# Review the schema of your data
df.printSchema()
root
|-- step: integer (nullable = true)
|-- type: string (nullable = true)
|-- amount: double (nullable = true)
|-- nameOrig: string (nullable = true)
|-- oldbalanceOrg: double (nullable = true)
|-- newbalanceOrig: double (nullable = true)
|-- nameDest: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

Types of transactions

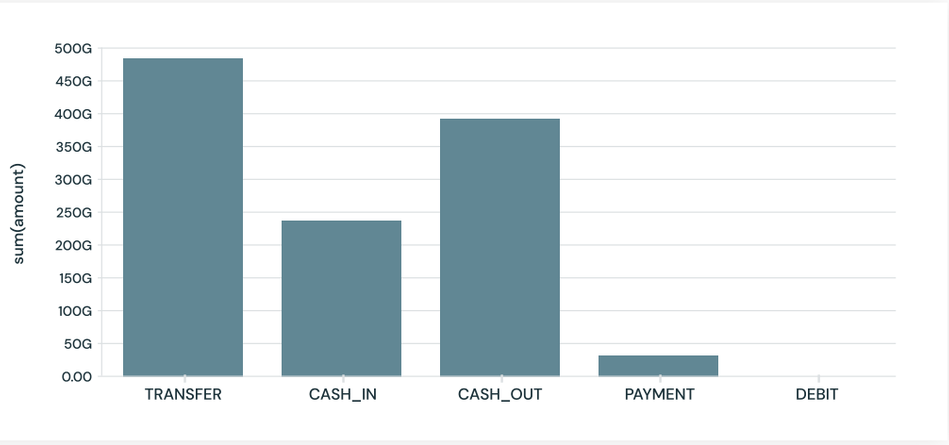
Let's visualize the data to understand the types of transactions the data captures and their contribution to the overall transaction volume.

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



To get an idea of how much money we are talking about, let's also visualize the data based on the types of transactions and on their contribution to the amount of cash transferred (i.e., sum(amount)).

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



Rules-based model

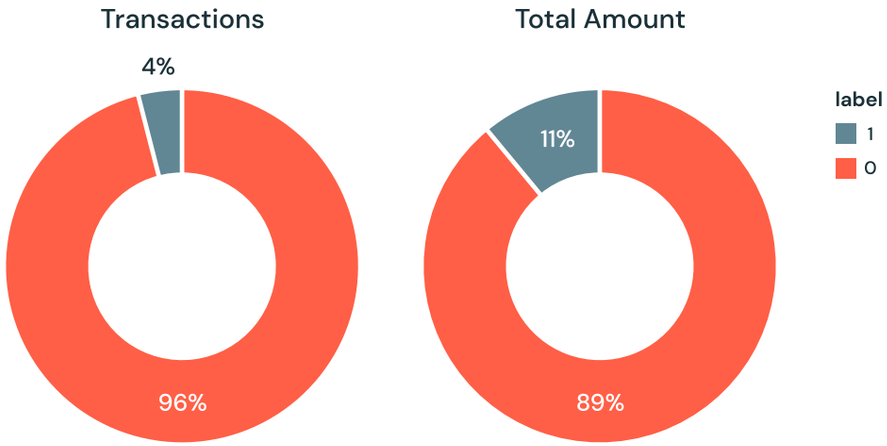
We are not likely to start with a large data set of known fraud cases to train our model. In most practical applications, fraudulent detection patterns are identified by a set of rules established by the domain experts. Here, we create a column called "label" based on these rules.

```
# 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))
```

Visualizing data flagged by rules

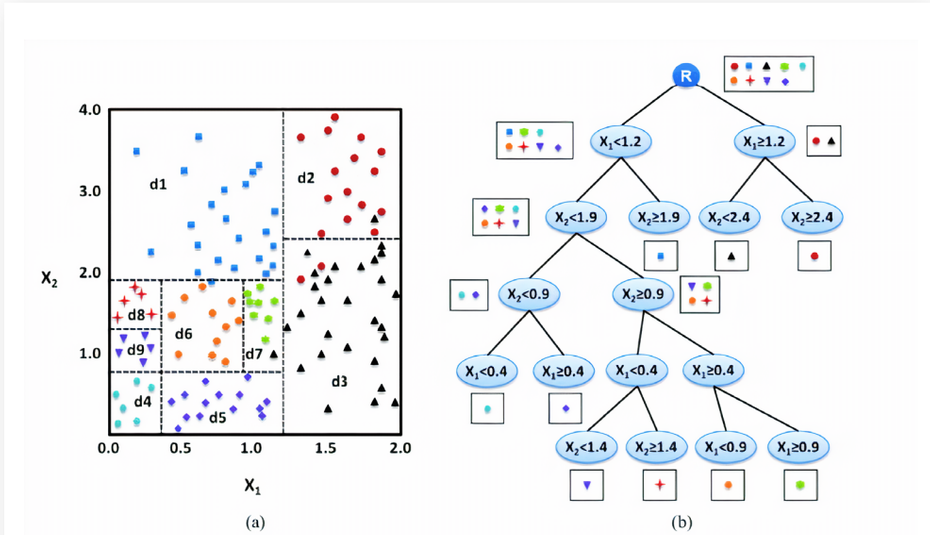
These rules often flag quite a large number of fraudulent cases. Let's visualize the number of flagged transactions. We can see that the rules flag about 4% of the cases and 11% of the total dollar amount as fraudulent.

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



Selecting the appropriate machine learning models

In many cases, a black box approach to fraud detection cannot be used. First, the domain experts need to be able to understand why a transaction was identified as fraudulent. Then, if action is to be taken, the evidence has to be presented in court. The decision tree is an easily interpretable model and is a great starting point for this use case. Read the blog "The wise old tree" on decision trees to learn more.



Creating the training set

To build and validate our ML model, we will do an 80/20 split using `.randomSplit`. This will set aside a randomly chosen 80% of the data for training and the remaining 20% to validate the results.

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

Creating the ML model pipeline

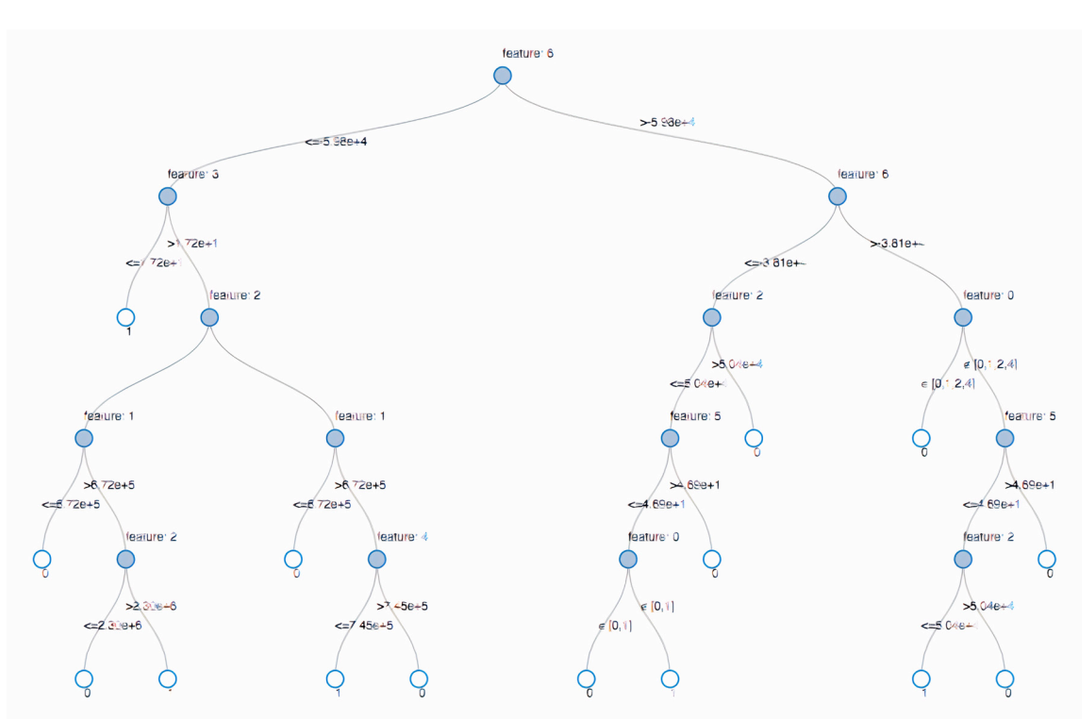
To prepare the data for the model, we must first convert categorical variables to numeric using `.StringIndexer`. We then must assemble all of the features we would like for the model to use. We create a pipeline to contain these feature preparation steps in addition to the decision tree model so that we may repeat these steps on different data sets. Note that we fit the pipeline to our training data first and will then use it to transform our test data in a later step.

```
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)
```

Visualizing the model

Calling `display()` on the last stage of the pipeline, which is the decision tree model, allows us to view the initial fitted model with the chosen decisions at each node. This helps us to understand how the algorithm arrived at the resulting predictions.

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



Visual representation of the Decision Tree model

Model tuning

To ensure we have the best fitting tree model, we will cross-validate the model with several parameter variations. Given that our data consists of 96% negative and 4% positive cases, we will use the Precision-Recall (PR) evaluation metric to account for the unbalanced distribution.

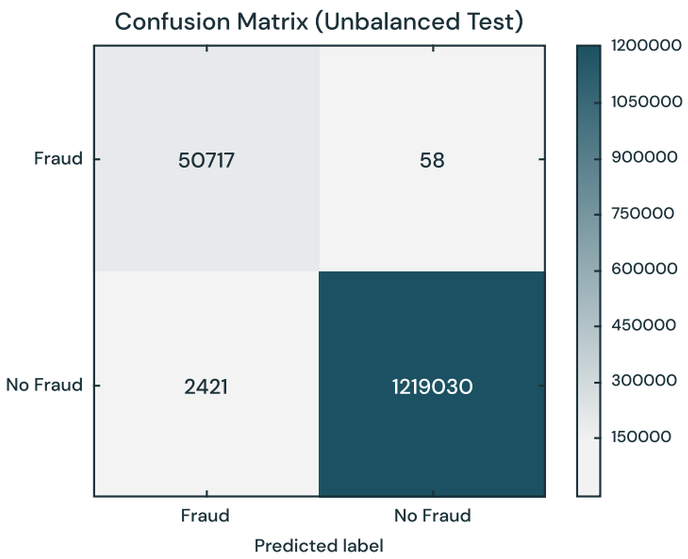
```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# 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)
```

Model performance

We evaluate the model by comparing the Precision-Recall (PR) and area under the ROC curve (AUC) metrics for the training and test sets. Both PR and AUC appear to be very high.

```
# 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
```

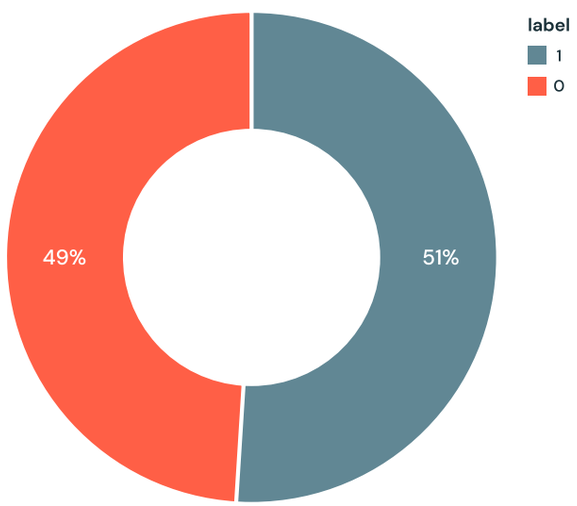
To see how the model misclassified the results, let's use Matplotlib and pandas to visualize our confusion matrix.



Balancing the classes

We see that the model is identifying 2,421 more cases than the original rules identified. This is not as alarming, as detecting more potential fraudulent cases could be a good thing. However, there are 58 cases that were not detected by the algorithm but were originally identified. We are going to attempt to improve our prediction further by balancing our classes using undersampling. That is, we will keep all the fraud cases and then downsample the non-fraud cases to match that number to get a balanced data set. When we visualize our new data set, we see that the yes and no cases are 50/50.

```
## 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())
```



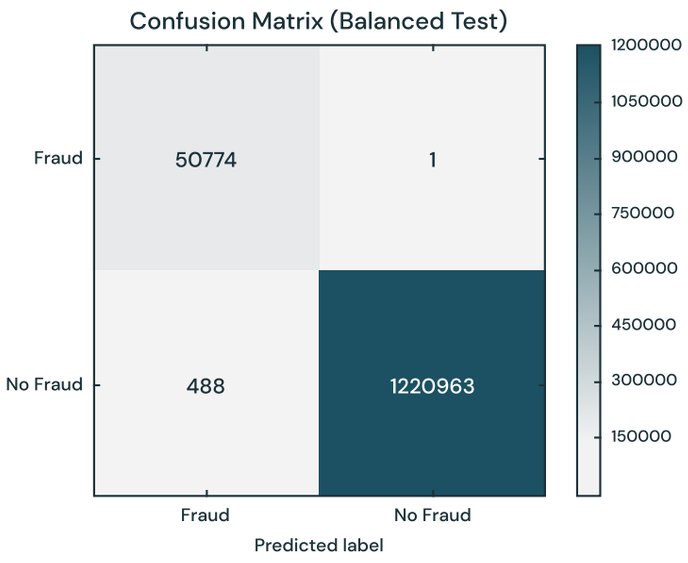
Updating the pipeline

Now let's update the ML pipeline and create a new cross validator. Because we are using ML pipelines, we only need to update it with the new data set and we can quickly repeat the same pipeline steps.

```
# 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
```

Review the results

Now let's look at the results of our new confusion matrix. The model misidentified only one fraudulent case. Balancing the classes seems to have improved the model.



Model feedback and using MLflow

Once a model is chosen for production, we want to continuously collect feedback to ensure that the model is still identifying the behavior of interest. Since we are starting with a rule-based label, we want to supply future models with verified true labels based on human feedback. This stage is crucial for maintaining confidence and trust in the machine learning process. Since analysts are not able to review every single case, we want to ensure we are presenting them with carefully chosen cases to validate the model output. For example, predictions, where the model has low certainty, are good candidates for analysts to review. The addition of this type of feedback will ensure the models will continue to improve and evolve with the changing landscape.

MLflow helps us throughout this cycle as we train different model versions. We can keep track of our experiments, comparing the results of different model configurations and parameters. For example, here we can compare the PR and AUC of the models trained on balanced and unbalanced data sets using the MLflow UI. Data scientists can use MLflow to keep track of the various model metrics and any additional visualizations and artifacts to help make the decision of which model should be deployed in production. The data engineers will then be able to easily retrieve the chosen model along with the library versions used for training as a .jar file to be deployed on new data in production. Thus, the collaboration between the domain experts who review the model results, the data scientists who update the models, and the data engineers who deploy the models in production will be strengthened throughout this iterative process.

Conclusion

We have reviewed an example of how to use a rule-based fraud detection label and convert it to a machine learning model using Databricks with MLflow. This approach allows us to build a scalable, modular solution that will help us keep up with ever-changing fraudulent behavior patterns. Building a machine learning model to identify fraud allows us to create a feedback loop that allows the model to evolve and identify new potential fraudulent patterns. We have seen how a decision tree model, in particular, is a great starting point to introduce machine learning to a fraud detection program due to its interpretability and excellent accuracy.

A major benefit of using the Databricks platform for this effort is that it allows for data scientists, engineers and business users to seamlessly work together throughout the process. Preparing the data, building models, sharing the results and putting the models into production can now happen on the same platform, allowing for unprecedented collaboration. This approach builds trust across the previously siloed teams, leading to an effective and dynamic fraud detection program.

Try this notebook by signing up for a free trial in just a few minutes and get started creating your own models.

CHAPTER 8:

How Virgin Hyperloop One Reduced Processing Time From Hours to Minutes With Koalas

A field guide to seamlessly switching your pandas code to Apache Spark™

By **Patryk Oleniuk** and **Sandhya Raghavan**

August 22, 2019

At Virgin Hyperloop One, we work on making hyperloop a reality, so we can move passengers and cargo at airline speeds but at a fraction of the cost of air travel. In order to build a commercially viable system, we collect and analyze a large, diverse quantity of data, including Devloop Test Track runs, numerous test rigs, and various simulation, infrastructure and socioeconomic data. Most of our scripts handling that data are written using Python libraries with pandas as the main data processing tool that glues everything together. In this blog post, we want to share with you our experiences of scaling our data analytics using Koalas, achieving massive speedups with minor code changes.

As we continue to grow and build new stuff, our data-processing needs grow too. Due to the increasing scale and complexity of our data operations, our pandas-based Python scripts were too slow to meet our business needs. This led us to Spark, with the hopes of fast processing times and flexible data storage as well as on-demand scalability. We were, however, struggling with the "Spark switch" — we would have to make a lot of custom changes to migrate our pandas-based code base to PySpark. We needed a solution that was not only much faster, but also would ideally not require rewriting code. These challenges drove us to research other options, and we were very happy to discover that there exists an easy way to skip that tedious step: the Koalas package, recently open sourced by Databricks.

As described in the Koalas Readme,

> The Koalas project makes data scientists more productive when interacting with big data, by implementing the pandas DataFrame API on top of Apache Spark.
>
> (...)
>
> Be immediately productive with Spark, with no learning curve, if you are already familiar with pandas.
>
> **Have a single code base that works both with pandas (tests, smaller data sets) and with Spark (distributed data sets).**

In this article I will try to show that this is (mostly) true and why Koalas is worth trying out. By making changes to less than 1% of our pandas lines, we were able to run our code with Koalas and Spark. We were able to reduce the execution times by more than 10x, from a few hours to just a few minutes, and since the environment is able to scale horizontally, we're prepared for even more data.

Quick start

Before installing Koalas, make sure that you have your Spark cluster set up and can use it with PySpark. Then, simply run:

```
pip install koalas
```

or, for conda users:

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

Refer to Koalas Readme for more details.

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

As you can see, Koalas can render pandas-like interactive tables. How convenient.

Example with basic operations

For the sake of this article, we generated some test data consisting of four columns and parameterized number of rows.

```
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
```

Disclaimer: This is a randomly generated test file used for performance evaluation, related to the topic of hyperloop, but not representing our data. The full test script used for this article can be found here.

We'd like to assess some key descriptive analytics across all pod-trips — for example: What is the trip time per pod-trip?

Operations needed:

1. Group by `['pod_id','trip_id']`
2. For every trip, calculate the `trip_time` as last timestamp — first timestamp
3. Calculate distribution of the pod-trip times (mean, stddev)

The short and slow (pandas) way (snippet #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()
```

The long and fast (PySpark) way (snippet #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()
```

The short and fast (Koalas) way (snippet #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()
```

Note that for the snippets #1 and #3, the code is exactly the same, and so the "Spark switch" is seamless. For most of the pandas scripts, you can even try to change the import pandas databricks.koalas as pd, and some scripts will run fine with minor adjustments, with some limitations explained below.

Results

All the snippets have been verified to return the same pod-trip-times results. The describe and summary methods for pandas and Spark are slightly different, as explained here but this should not affect performance too much.

Sample results:

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

Advanced example: UDFs and complicated operations

We're now going to try to solve a more complex problem with the same DataFrame and see how pandas and Koalas implementations differ.

Goal: Analyze the average speed per pod-trip:

1. Group by `['pod_id','trip id']`
2. For every pod-trip, calculate the total distance travelled by finding the area below the velocity (time) chart (method explained here)
3. Sort the grouped df by `timestamp` column
4. Calculate diffs of timestamps
5. Multiply the diffs with the speed — this will result in the distance travelled in that time diff
6. Sum the `distance_travelled` column — this will give us total distance travelled per pod-trip
7. Calculate the `trip time` as `timestamp.last` – `timestamp.first` (as in the previous paragraph)
8. Calculate the `average_speed` as `distance_travelled` / `trip time`
9. Calculate distribution of the pod-trip times (mean, stddev)

We decided to implement this task using a custom apply function and user-defined functions (UDF).

The pandas way (snippet #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()
```

The PySpark way (snippet #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
```

The Koalas way (snippet #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()
```

Koalas' implementation of apply is based on PySpark's `pandas_udf`, which requires schema information, and this is why the definition of the function has to also define the type hint. The authors of the package introduced new custom type hints, `ks.DataFrame` and `ks.Series`. Unfortunately, the current implementation of the apply method is quite cumbersome, and it took a bit of an effort to arrive at the same result (column names change, groupBy keys not returned). However, all the behaviors are appropriately explained in the package documentation.

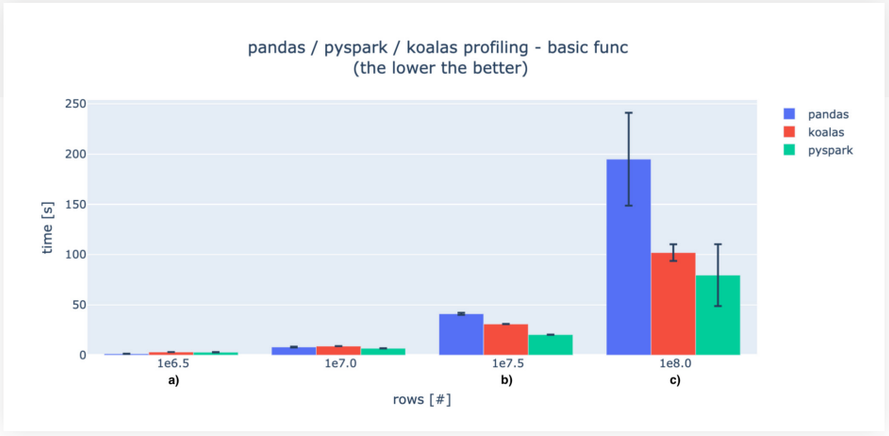
Performance

To assess the performance of Koalas, we profiled the code snippets for a different number of rows.

The profiling experiment was done on the Databricks platform, using the following cluster configurations:

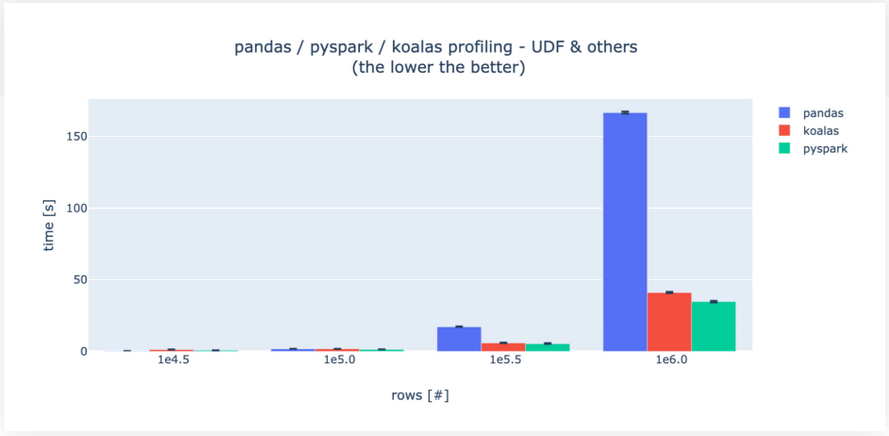
- Spark driver node (also used to execute the pandas scripts): 8 CPU cores, 61GB RAM
- 15 Spark worker nodes: 4 CPU cores, 30.5GB RAM each (sum: 60CPUs / 457.5GB)

Every experiment was repeated 10 times, and the clips shown below are indicating the min and max times for the executions.



Basic ops

When the data is small, the initialization operations and data transfer are huge in comparison to the computations, so pandas is much faster (marker a). For larger amounts of data, pandas' processing times exceed distributed solutions (marker b). We can then observe some performance hits for Koalas, but it gets closer to PySpark as data increases (marker c).



UDFs

For the UDF profiling, as specified in PySpark and Koalas documentation, the performance decreases dramatically. This is why we needed to decrease the number of rows we tested with by 100x vs the basic ops case. For each test case, Koalas and PySpark show a striking similarity in performance, indicating a consistent underlying implementation. During experimentation, we discovered that there exists a much faster way of executing that set of operations using PySpark windows functionality; however, this is not currently implemented in Koalas so we decided to only compare UDF versions.

Discussion

Koalas seems to be the right choice if you want to make your pandas code immediately scalable and executable on bigger data sets that are not possible to process on a single node. After the quick swap to Koalas, just by scaling your Spark cluster, you can allow bigger data sets and improve the processing times significantly. Your performance should be comparable (but 5% to 50% lower, depending on the data set size and the cluster) with PySpark's.

On the other hand, the Koalas API layer does cause a visible performance hit, especially in comparison to the native Spark. At the end of the day, if computational performance is your key priority, you should consider switching from Python to Scala.

Limitations and differences

During your first few hours with Koalas, you might wonder, "Why is this not implemented?!" Currently, the package is still under development and is missing some pandas API functionality, but much of it should be implemented in the next few months (for example `groupby.diff()` or `kdf.rename()`).

Also from my experience as a contributor to the project, some of the features are either too complicated to implement with Spark API or were skipped due to a significant performance hit. For example, DataFrame.values requires materializing the entire working set in a single node's memory, and so is suboptimal and sometimes not even possible. Instead, if you need to retrieve some final results on the driver, you can call `DataFrame.to_pandas()` or `DataFrame.to_numpy()`.

Another important thing to mention is that Koalas' execution chain is different from pandas': When executing the operations on the DataFrame, they are put on a queue of operations but **not** executed. Only when the results are needed — e.g., when calling `kdf.head()` or `kdf.to_pandas()` — will the operations be executed. That could be misleading for somebody who never worked with Spark, since pandas does everything line by line.

Conclusions

Koalas helped us to reduce the burden to "Spark-ify" our pandas code. If you're also struggling with scaling your pandas code, you should try it too. If you are desperately missing any behavior or found inconsistencies with pandas, please open an issue so that as a community we can ensure that the package is actively and continually improved. Also, feel free to contribute.

Resources

1. Koalas GitHub
2. Koalas documentation
3. Code snippets from this article

> Start experimenting with these free Databricks notebooks (pandas vs. Koalas).

CHAPTER 9:

Delivering a Personalized Shopping Experience With Apache Spark™ on Databricks

By **Brett Bevers**

March 31, 2017

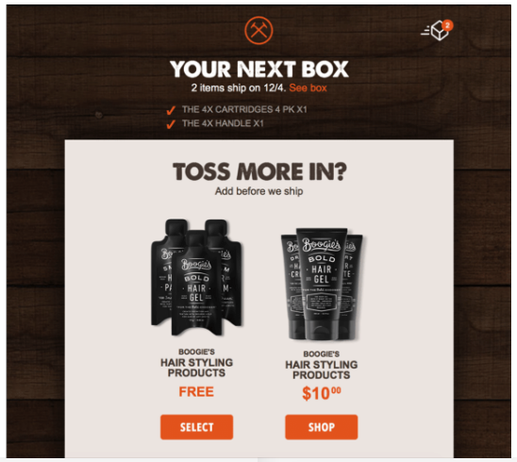
Dollar Shave Club (DSC) is a men's lifestyle brand and e-commerce company on a mission to change the way men address their shaving and grooming needs. Data is perhaps the most critical asset in achieving a cutting-edge user experience. Databricks has been an important partner in our efforts to build a personalized customer experience through data. This post describes how the Databricks platform supported all stages of development and deployment of a powerful, custom machine learning pipeline.

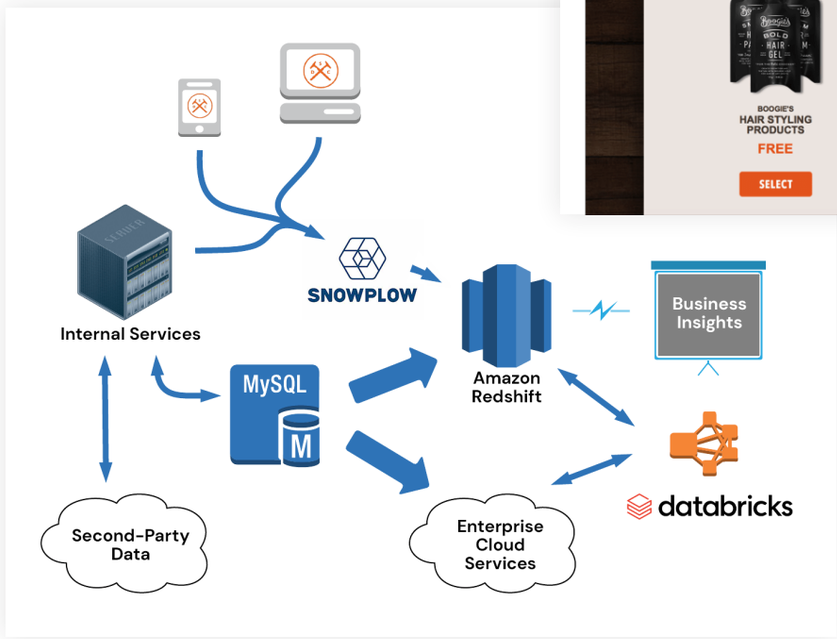
DSC's primary offering is a monthly subscription for razor cartridges, which are shipped directly to members. Our members join and manage their account on our single-page web app or native mobile apps. During their visit, they can shop our catalogue of grooming and bathroom products — we now have dozens of products organized under distinctive brands. Courtesy of the club, members and guests can enjoy original content, articles and videos created for people who enjoy our characteristic style. They can satisfy their curiosity on health and grooming topics with articles that don't remind them of their junior high health class. They can get quick tips on style, work and relationships, or they can read DSC's fun take on big questions, like "How long can civilization last on Earth?" DSC also seeks to engage people on social media channels, and our members can be enthusiastic about joining in. By identifying content and offers of the most interest to each individual member, we can provide a more personalized and better membership experience.

Data at Dollar Shave Club

DSC's interactions with members and guests generate a mountain of data. Knowing that the data would be an asset in improving member experience, our engineering team invested early in a modern data infrastructure. Our web applications, internal services and data infrastructure are 100% hosted on AWS. A Redshift cluster serves as the central data warehouse, receiving data from various systems. Records are continuously replicated from production databases into the warehouse. Data also moves between applications and into Redshift mediated by Apache Kafka, an open source streaming platform. We use Snowplow, a highly customizable open source event pipeline, to collect event data from our web and mobile clients as well as server-side applications. Clients emit detailed records of page views, link clicks, browsing activity and any number of custom events and contexts. Once data reaches Redshift, it is accessed through various analytics platforms for monitoring, visualization and insights.

With this level of visibility, there are abundant opportunities to learn from our data and act on it. But identifying those opportunities and executing at scale requires the right tools. Apache Spark — a state-of-the-art cluster computing framework with engines for ETL, stream processing and machine learning — is an obvious choice. Moreover, Databricks latest developments for data engineering make it exceedingly easy to get started with Spark, providing a platform that is apt as both an IDE and deployment pipeline. On our first day using Databricks, we were equipped to grapple with a new class of data challenges.





Use case: Recommendation engine

One of the first projects we developed on Databricks aimed to use predictive modeling to optimize the product recommendations that we make to our members over a particular email channel. Members receive a sequence of emails in the week before their subscription box is shipped. These emails inform them about the upcoming shipment and also suggest additional products that they can include in their box. Members can elect to add a recommended product from the email with just a few clicks. Our goal was to produce, for a given member, a ranking of products that prescribes which products to promote in their monthly email and with what priority.

We planned to perform an exhaustive exploration in search of behavior that tends to indicate a member's level of interest in each of our products. We would extract a variety of metrics in about a dozen segments of member data, pivot that data by hundreds of categories, actions and tags, and index event-related metrics by discretized time. In all, we included nearly 10,000 features, for a large cohort of members, in the scope of our exploration. To contend with a large, high-dimensional and sparse data set, we decided to automate the required ETL and data mining techniques using Spark Core, Spark SQL and MLlib. The final product would be a collection of linear models, trained and tuned on production data, that could be combined to produce product rankings.

We set out to develop a fully automated pipeline on Spark with the following stages:

1. Extract data from warehouse (Redshift)
2. Aggregate and pivot data per member
3. Select features to include in final models

Step 1: Extract data

We start by looking at various segments of data in our relational databases; groupings of records that need to be stitched together to describe a domain of events and relationships. We need to understand each data segment — how it can be interpreted and how it might need to be cleaned up — so that we can extract accurate, portable, self-describing representations of the data. This is crucial work that collects together domain expertise and institutional knowledge about the data and its lifecycle, so it is important to document and communicate what is learned in the process. Databricks provides a "notebook" interface to the Spark shell that makes it easy to work with data interactively, while having complete use of Spark's programming model. Spark notebooks proved to be ideal for trying out ideas and quickly sharing the results or keeping a chronicle of your work for reference later.

For each data segment, we encapsulate the specifics of cleaning and denormalizing records in an extractor module. In many cases, we can simply export tables from Redshift, dynamically generate SQL queries, and then let Spark SQL do all the heavy lifting. If necessary, we can cleanly introduce functional programming using Spark's DataFrames API. And the application of domain-specific metadata has a natural home in an extractor. Importantly, the first steps for processing a particular data segment are neatly isolated from that for other segments and from other stages of the pipeline. Extractors can be developed and tested independently. And they can be reused for other explorations or for production pipelines.

```
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)
```

Example code for a data extraction pipeline. The pipeline uses the interface implemented by several extractor classes, passing arguments to customize behavior. The pipeline is agnostic to the details of each extraction.

```
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()
```

Example code for an extractor interface. In many cases, an extractor simply generates a SQL query and passes it to Spark SQL.

Step 2: Aggregate and pivot

The data extracted from the warehouse is mostly detailed information about individual events and relationships. But what we really need is an aggregated description of activity over time, so that we can effectively search for behavior that indicates interest in one product or another. Comparing specific event types second by second is not fruitful — at that level of granularity, the data is much too sparse to be good fodder for machine learning. The first thing to do is to aggregate event-related data over discrete periods of time. Reducing events into counts, totals, averages, frequencies, etc., makes comparisons among members more meaningful and it makes data mining more tractable. Of course, the same set of events can be aggregated over several different dimensions, in addition to time. Any or all of these numbers could tell an interesting story.

Aggregating over each of several attributes in a data set is often called pivoting (or rolling up) the data. When we group data by member and pivot by time and other interesting attributes, the data is transformed from data about individual events and relationships into a (very long) list of features that describe our members. For each data segment, we encapsulate the specific method for pivoting the data in a meaningful way in its own module. We call these modules transformers. Because the pivoted data set can be extremely wide, it is often more performant to work with RDDs rather than DataFrames. We generally represent the set of pivoted features using a sparse vector format, and we use KeyValue RDD transformations to reduce the data. Representing member behavior as a sparse vector shrinks the size of the data set in memory and also makes it easy to generate training sets for use with MLlib in the next stage of the pipeline.

Step 3: Data mining

At this point in the pipeline, we have an extremely large set of features for each member and we want to determine, for each of our products, which subset of those features best indicate when a member is interested in purchasing that product. This is a data mining problem. If there were fewer features to consider — say, dozens instead of several thousand — then there would be several reasonable ways to proceed. However, the large number of features being considered presents a particularly hard problem. Thanks to the Databricks platform, we were easily able to apply a tremendous amount of computing time to the problem. We used a method that involved training and evaluating models on relatively small, randomly sampled sets of features. Over several hundred iterations, we gradually accumulate a subset of features that each make a significant contribution to a high-performing model. Training a model and calculating the evaluation statistic for each feature in that model is computationally expensive. But we had no trouble provisioning large Spark clusters to do the work for each of our products and then terminating them when the job was done.

It is essential that we be able to monitor the progress of the data-mining process. If there is a bug or data quality issue that is preventing the process from converging on the best-performing model, then we need to detect it as early as possible to avoid wasting many hours of processing time. For that purpose, we developed a simple dashboard on Databricks that visualizes the evaluation statistics collected on each iteration.

Final models

The evaluation module in MLlib makes it exceedingly simple to tune the parameters of its models. Once the hard work of the ETL and data-mining process is complete, producing the final models is nearly effortless. After determining the final model coefficients and parameters, we were prepared to start generating product rankings in production. We used Databricks scheduling feature to run a daily job to produce product rankings for each of the members who will be receiving an email notification that day. To generate feature vectors for each member, we simply apply to the most recent data the same extractor and transformer modules that generated the original training data. This not only saved development time up front, it avoids the problems of dual maintenance of exploration and production pipelines. It also ensures that the models are being applied under the most favorable conditions — to features that have precisely the same meaning and context as in the training data.

Future plans with Databricks and Apache Spark

The product recommendation project turned out to be a great success that has encouraged us to take on similarly ambitious data projects at DSC. Databricks continues to play a vital role in supporting our development workflow, particularly for data products. Large-scale data mining has become an essential tool that we use to gather information to address strategically important questions, and the resulting predictive models can then be deployed to power smart features in production. In addition to machine learning, we have projects that employ stream-processing applications built on Spark Streaming; for example, we consume various event streams to unobtrusively collect and report metrics, or to replicate data across systems in near real-time. And, of course, a growing number of our ETL processes are being developed on Spark.

CHAPTER 10:

Parallelizing Large Simulations With Apache SparkR on Databricks

By **Wayne W. Jones**, **Dennis Vallinga** and **Hossein Falaki**

June 23, 2017

Introduction

Apache Spark 2.0 introduced a new family of APIs in SparkR, the R interface to Apache Spark™, to enable users to parallelize existing R functions. The new dapply, gapply and spark.lapply methods open exciting possibilities for R users. In this post, we present details on one use case jointly done by Shell Oil Company and Databricks.

Use case: Stocking recommendation

In Shell, current stocking practices are often driven by a combination of vendor recommendations, prior operational experience and "gut feeling." As such, a limited focus is directed toward incorporating historical data in these decisions, which can sometimes lead to excessive or insufficient stock being held at Shell's locations (e.g., an oil rig).

The prototype tool, Inventory Optimization Analytics solution, has proven that Shell can use advanced data analysis techniques on SAP inventory data to:

- Optimize warehouse inventory levels
- Forecast safety stock levels
- Rationalize slow-moving materials
- Review and reassign non-stock and stock items on materials list
- Identify material criticality (e.g., via bill of materials linkage, past usage or lead time)

To calculate the recommended stocking inventory level requirement for a material, the data science team has implemented a Markov Chain Monte Carlo (MCMC) bootstrapping statistical model in R. The model is applied to each and every material (typically 3,000-plus) issued across 50-plus Shell locations. Each individual material model involves simulating 10,000 MCMC iterations to capture the historical distribution of issues. Cumulatively, the computational task is large but, fortunately, is one of an embarrassingly parallel nature because the model can be applied independently to each material.

Existing setup

The full model is currently executed on a 48-core, 192GB RAM stand-alone physical offline PC. The MCMC bootstrap model is a custom-built set of functions that use a number of third-party R packages:

```
("fExtremes", "ismev", "dplyr", "tidyr", "stringr")
```

The script iterates through each of the Shell locations and distributes the historical material into roughly equally sized groups of materials across the 48 cores. Each core then iteratively applies the model to each individual material. We are grouping the materials because a simple loop for each material would create too much overhead (e.g., starting the R process, etc.) as each calculation takes 2-5 seconds. The distribution of the material group jobs across the cores is implemented via the R parallel package. When the last of the individual 48 core jobs complete, the script moves on to the next location and repeats the process. The script takes a total time of approximately 48 hours to calculate the recommended inventory levels for all Shell locations.

Using Apache Spark on Databricks

Instead of relying on a single large machine with many cores, Shell decided to use cluster computing to scale out. The new R API in Apache Spark was a good fit for this use case. Two versions of the workload were developed as prototypes to verify scalability and performance of SparkR.

Prototype I: A proof of concept

For the first prototype, we tried to minimize the amount of code change, as the goal was to quickly validate that the new SparkR API can handle the workload. We limited all the changes to the simulation step as follows:

For each Shell location list element:

1. Parallelize input date as a Spark DataFrame
2. Use `SparkR::gapply()` to perform parallel simulation for each of the chunks

With limited change to the existing simulation code base, we could reduce the total simulation time to 3.97 hours on a 50-node Spark cluster on Databricks.

Prototype II: Improving performance

While the first prototype was quick to implement, it suffered from one obvious performance bottleneck: A Spark job is launched for every iteration of the simulation. The data is highly skewed and as a result, during each job most executors are waiting idle for the stragglers to finish before they can take on more work from the next job. Further, at the beginning of each job, we spend time parallelizing the data as a Spark DataFrame while most of the CPU cores on the cluster are idle.

To solve these problems, we modified the preprocessing step to produce input and auxiliary data for all locations and material values up front. Input data was parallelized as a large Spark DataFrame. Next, we used a single `SparkR::gapply()` call with two keys: location ID and material ID to perform the simulation.

With these simple improvements, we could reduce the simulation time to 45 minutes on a 50-node Spark cluster on Databricks.

Improvements to SparkR

SparkR is one of the latest additions to Apache Spark, and the apply API family was the latest addition to SparkR at the time of this work. Through this experiment, we identified a number of limitations and bugs in SparkR and fixed them in Apache Spark.

- [SPARK-17790] Support for parallelizing R data.frame larger than 2GB
- [SPARK-17919] Make timeout to RBackend configurable in SparkR
- [SPARK-17811] SparkR cannot parallelize data.frame with NA or NULL in Date columns

What's next?

If you are a SparkR developer and you want to explore SparkR, get an account on Databricks today and peruse our SparkR documentation.

CHAPTER 11: CUSTOMER CASE STUDIES

Comcast delivers the future of entertainment

"With Databricks, we can now be more informed about the decisions we make, and we can make them faster."

— **Jim Forsythe**
Senior Director, Product Analytics and Behavioral Sciences
Comcast

As a global technology and media company that connects millions of customers to personalized experiences, Comcast struggled with massive data, fragile data pipelines and poor data science collaboration. By using Databricks — including Delta Lake and MLflow — they were able to build performant data pipelines for petabytes of data and easily manage the lifecycle of hundreds of models, creating a highly innovative, unique and award-winning viewer experience that leverages voice recognition and machine learning.

Use case: In the intensely competitive entertainment industry, there's no time to press the Pause button. Comcast realized they needed to modernize their entire approach to analytics, from data ingest to the deployment of machine learning models that deliver new features to delight their customers.

Solution and benefits: Armed with a unified approach to analytics, Comcast can now fast-forward into the future of AI-powered entertainment — keeping viewers engaged and delighted with competition-beating customer experiences.

- **Emmy-winning viewer experience:** Databricks helps enable Comcast to create a highly innovative and award-winning viewer experience with intelligent voice commands that boost engagement
- **Reduced compute costs by 10x:** Delta Lake has enabled Comcast to optimize data ingestion, replacing 640 machines with 64 — while improving performance. Teams can spend more time on analytics and less time on infrastructure management.
- **Higher data science productivity:** The upgrades and use of Delta Lake fostered global collaboration among data scientists by enabling different programming languages through a single interactive workspace. Delta Lake also enabled the data team to use data at any point within the data pipeline, allowing them to act much quicker in building and training new models.
- **Faster model deployment:** By modernizing, Comcast reduced deployment times from weeks to minutes as operations teams deployed models on disparate platforms

Learn more

CHAPTER 11: CUSTOMER CASE STUDIES

Regeneron accelerates drug discovery with genomic sequencing

REGENERON

"The Databricks Unified Data Analytics Platform is enabling everyone in our integrated drug development process — from physician-scientists to computational biologists — to easily access, analyze and extract insights from all of our data."

— **Jeffrey Reid, Ph.D.**
Head of Genome Informatics
Regeneron

Regeneron's mission is to tap into the power of genomic data to bring new medicines to patients in need. Yet, transforming this data into life-changing discovery and targeted treatments has never been more challenging. With poor processing performance and scalability limitations, their data teams lacked what they needed to analyze petabytes of genomic and clinical data. Databricks now empowers them to quickly analyze entire genomic data sets quickly to accelerate the discovery of new therapeutics.

Use case: More than 95% of all experimental medicines that are currently in the drug development pipeline are expected to fail. To improve these efforts, the Regeneron Genetics Center built one of the most comprehensive genetics databases by pairing the sequenced exomes and electronic health records of more than 400,000 people. However, they faced numerous challenges analyzing this massive set of data:

- Genomic and clinical data is highly decentralized, making it very difficult to analyze and train models against their entire 10TB data set
- Difficult and costly to scale their legacy architecture to support analytics on over 80 billion data points
- Data teams were spending days just trying to ETL the data so that it can be used for analytics

Solution and benefits: Databricks provides Regeneron with a Unified Data Analytics Platform running on Amazon Web Services that simplifies operations and accelerates drug discovery through improved data science productivity. This is empowering them to analyze the data in new ways that were previously impossible.

- **Accelerated drug target identification:** Reduced the time it takes data scientists and computational biologists to run queries on their entire data set from 30 minutes down to 3 seconds — a 600x improvement!
- **Increased productivity:** Improved collaboration, automated DevOps and accelerated pipelines (ETL in 2 days vs 3 weeks) have enabled their teams to support a broader range of studies

Learn more

databricks

CHAPTER 11: CUSTOMER CASE STUDIES

Nationwide reinvents insurance with actuarial modeling

"With Databricks, we are able to train models against all our data more quickly, resulting in more accurate pricing predictions that have had a material impact on revenue."

— **Bryn Clark**
Data Scientist
Nationwide

The explosive growth in data availability and increasing market competition are challenging insurance providers to provide better pricing to their customers. With hundreds of millions of insurance records to analyze for downstream ML, Nationwide realized their legacy batch analysis process was slow and inaccurate, providing limited insight to predict the frequency and severity of claims. With Databricks, they have been able to employ deep learning models at scale to provide more accurate pricing predictions, resulting in more revenue from claims.

Use case: The key to providing accurate insurance pricing lies in leveraging information from insurance claims. However, data challenges were difficult, as they had to analyze insurance records that were volatile because claims were infrequent and unpredictable — resulting in inaccurate pricing.

Solution and benefits: Nationwide leverages the Databricks Unified Data Analytics Platform to manage the entire analytics process from data ingestion to the deployment of deep learning models. The fully managed platform has simplified IT operations and unlocked new data-driven opportunities for their data science teams.

- **Data processing at scale:** Improved runtime of their entire data pipeline from 34 hours to less than 4 hours, a 9x performance gain
- **Faster featurization:** Data engineering is able to identify features 15x faster — from 5 hours to around 20 minutes
- **Faster model training:** Reduced training times by 50%, enabling faster time-to-market of new models
- **Improved model scoring:** Accelerated model scoring from 3 hours to less than 5 minutes, a 60x improvement

CHAPTER 11: CUSTOMER CASE STUDIES

Condé Nast boosts reader engagement with experiences driven by data and AI

Condé Nast is one of the world's leading media companies, counting some of the most iconic magazine titles in its portfolio, including The New Yorker, Wired and Vogue. The company uses data to reach over 1 billion people in print, online, video and social media.

Use case: As a leading media publisher, Condé Nast manages over 20 brands in their portfolio. On a monthly basis, their web properties garner 100 million-plus visits and 800 million-plus page views, producing a tremendous amount of data. The data team is focused on improving user engagement by using machine learning to provide personalized content recommendations and targeted ads.

Solution and benefits: Databricks provides Condé Nast with a fully managed cloud platform that simplifies operations, delivers superior performance and enables data science innovation.

- **Improved customer engagement:** With an improved data pipeline, Condé Nast can make better, faster and more accurate content recommendations, improving the user experience
- **Built for scale:** Data sets can no longer outgrow Condé Nast's capacity to process and glean insights
- **More models in production:** With MLflow, Condé Nast's data science teams can innovate their products faster. They have deployed over 1,200 models in production.

Learn more

"Databricks has been an incredibly powerful end-to-end solution for us. It's allowed a variety of different team members from different backgrounds to quickly get in and utilize large volumes of data to make actionable business decisions."

— **Paul Fryzel**
Principal Engineer of AI Infrastructure
Condé Nast

CHAPTER 11: CUSTOMER CASE STUDIES

Showtime leverages ML to deliver data-driven content programming

"Being on the Databricks platform has allowed a team of exclusively data scientists to make huge strides in setting aside all those configuration headaches that we were faced with. It's dramatically improved our productivity."

— **Josh McNutt**
Senior Vice President of
Data Strategy and Consumer Analytics
Showtime

SHOWTIME® is a premium television network and streaming service, featuring award-winning original series and original limited series like "Shameless," "Homeland," "Billions," "The Chi," "Ray Donovan," "SMILF," "The Affair," "Patrick Melrose," "Our Cartoon President," "Twin Peaks" and more.

Use case: The Data Strategy team at Showtime is focused on democratizing data and analytics across the organization. They collect huge volumes of subscriber data (e.g., shows watched, time of day, devices used, subscription history, etc.) and use machine learning to predict subscriber behavior and improve scheduling and programming.

Solution and benefits: Databricks has helped Showtime democratize data and machine learning across the organization, creating a more data-driven culture.

- **6x faster pipelines:** Data pipelines that took over 24 hours are now run in less than 4 hours, enabling teams to make decisions faster
- **Removing infrastructure complexity:** Fully managed platform in the cloud with automated cluster management allows the data science team to focus on machine learning rather than hardware configurations, provisioning clusters, debugging, etc.
- **Innovating the subscriber experience:** Improved data science collaboration and productivity has reduced time-to-market for new models and features. Teams can experiment faster, leading to a better, more personalized experience for subscribers.

Learn more

CHAPTER 11: CUSTOMER CASE STUDIES

Shell innovates with energy solutions for a cleaner world

Shell is a recognized pioneer in oil and gas exploration and production technology and is one of the world's leading oil and natural gas producers, gasoline and natural gas marketers and petrochemical manufacturers.

Use case: To maintain production, Shell stocks over 3,000 different spare parts across their global facilities. It's crucial the right parts are available at the right time to avoid outages, but equally important is not overstocking, which can be cost-prohibitive.

Solution and benefits: Databricks provides Shell with a cloud-native unified analytics platform that helps with improved inventory and supply chain management.

- **Predictive modeling:** Scalable predictive model is developed and deployed across more than 3,000 types of materials at 50-plus locations
- **Historical analyses:** Each material model involves simulating 10,000 Markov Chain Monte Carlo iterations to capture historical distribution of issues
- **Massive performance gains:** With a focus on improving performance, the data science team reduced the inventory analysis and prediction time to 45 minutes from 48 hours on a 50-node Apache Spark™ cluster on Databricks — a 32x performance gain
- **Reduced expenditures:** Cost savings equivalent to millions of dollars per year

Learn more

"Databricks has produced an enormous amount of value for Shell. The inventory optimization tool [built on Databricks] was the first scaled-up digital product that came out of my organization, and the fact that it's deployed globally means we're now delivering millions of dollars of savings every year."

— **Daniel Jeavons**
General Manager of Advanced Analytics CoE
Shell

databricks

CHAPTER 11: CUSTOMER CASE STUDIES

Riot Games leverages AI to engage gamers and reduce churn

Riot Games' goal is to be the world's most player-focused gaming company. Founded in 2006 and based in LA, Riot Games is best known for the League of Legends game. Over 100 million gamers play every month.

Use case: Improving gaming experience through network performance monitoring and combating in-game abusive language.

Solution and benefits: Databricks allows Riot Games to improve the gaming experience of their players by providing scalable, fast analytics.

- **Improved in-game purchase experience:** Able to rapidly build and productionize a recommendation engine that provides unique offers based on over 500B data points. Gamers can now more easily find the content they want.
- **Reduced game lag:** Built ML model that detects network issues in real time, enabling Riot Games to avoid outages before they adversely impact players
- **Faster analytics:** Increased processing performance of data preparation and exploration by 50% compared to EMR, significantly speeding up analyses

Learn more

"We wanted to free data scientists from managing clusters. Having an easy-to-use, managed Spark solution in Databricks allows us to do this. Now our teams can focus on improving the gaming experience."

— **Colin Borys**
Data Scientist
Riot Games

CHAPTER 11: CUSTOMER CASE STUDIES

Eneco uses ML to reduce energy consumption and operating costs

"Databricks, through the power of Delta Lake and structured streaming, allows us to deliver alerts and recommendations to our customers with a very limited latency, so they're able to react to problems or make adjustments within their home before it affects their comfort levels."

— **Stephen Galsworthy**
Head of Data Science
Eneco

Eneco is the technology company behind Toon, the smart energy management device that gives people control over their energy usage, their comfort, the security of their homes and much more. Eneco's smart devices are in hundreds of thousands of homes across Europe. As such, they maintain Europe's largest energy data set, consisting of petabytes of IoT data, collected from sensors on appliances throughout the home. With this data, they are on a mission to help their customers live more comfortable lives while reducing energy consumption through personalized energy usage recommendations.

Use case: Personalized energy use recommendations: Leverage machine learning and IoT data to power their Waste Checker app, which provides personalized recommendations to reduce in-home energy consumption.

Solution and benefits: Databricks provides Eneco with a unified data analytics platform that has fostered a scalable and collaborative environment across data science and engineering, allowing data teams to more quickly innovate and deliver ML-powered services to Eneco's customers.

- **Lowered costs:** Cost-saving features provided by Databricks (such as autoscaling clusters and Spot instances) have helped Eneco significantly reduce the operational costs of managing infrastructure, while still being able to process large amounts of data
- **Faster innovation:** With their legacy architecture, moving from proof of concept to production took over 12 months. Now with Databricks, the same process takes less than eight weeks. This enables Quby's data teams to develop new ML-powered features for their customers much faster.
- **Reduced energy consumption:** Through their Waste Checker app, Eneco has identified over 67 million kilowatt hours of energy that can be saved by leveraging their personalized recommendations

Learn more

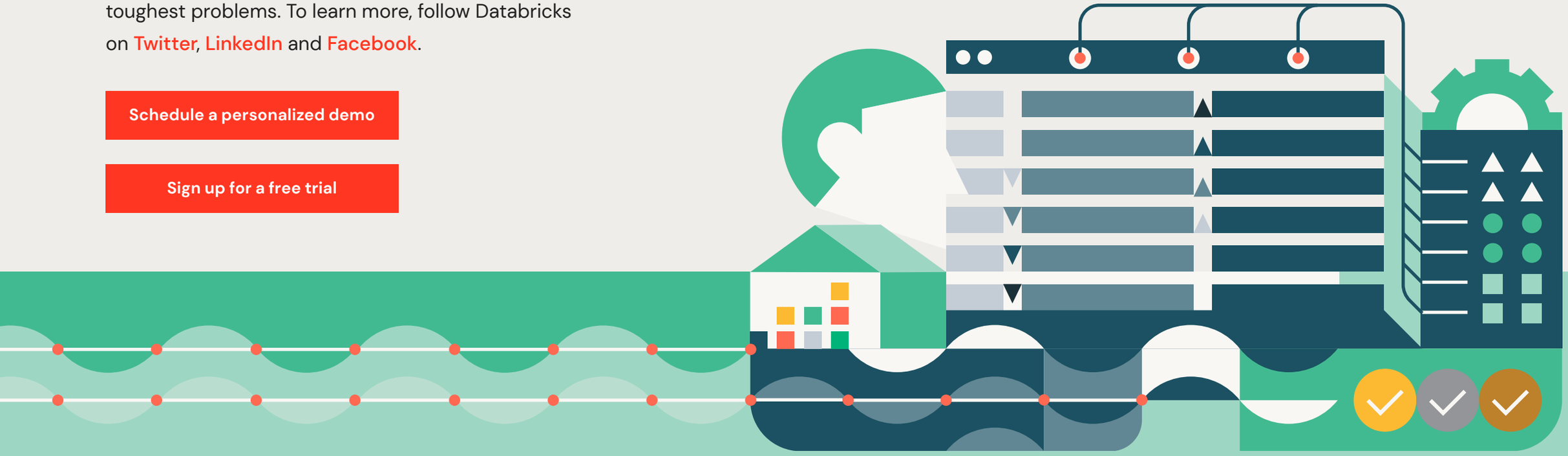
databricks

About Databricks

Databricks is the data and AI company. More than 5,000 organizations worldwide — including Comcast, Condé Nast, H&M and over 40% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark™, Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.

Schedule a personalized demo

Sign up for a free trial



databricks