2ND EDITION

eBook

Big Book of Machine Learning Use Cases

A collection of technical blogs, including code samples and notebooks

databricks

Contents

CHAPTER 1:

Introduction

Organizations across many industries are using machine learning to power new customer experiences, optimize business processes and improve employee productivity. From detecting financial fraud to improving the play-by-play decision-making for professional sports teams, this book brings together a multitude of practical use cases to get you started on your machine learning journey. The collection also serves as a guide — including code samples and notebooks — so you can roll up your sleeves and dive into machine learning on the Databricks Lakehouse.

CHAPTER 2:

Moneyball 2.0: Improving Pitch-by-Pitch Decision-Making With MLB's Statcast Data

By **Max Wittenberg**

Introduction

The Oakland Athletics baseball team in 2002 used data analysis and quantitative modeling to identify undervalued players and create a competitive lineup on a limited budget. The book "Moneyball," written by Michael Lewis, highlighted the A's '02 season and gave an inside glimpse into how unique the team's strategic data modeling was for its time. Fast-forward 20 years — the use of data science and quantitative modeling is now a common practice among all sports franchises and plays a critical role in scouting, roster construction, game-day operations and season planning.

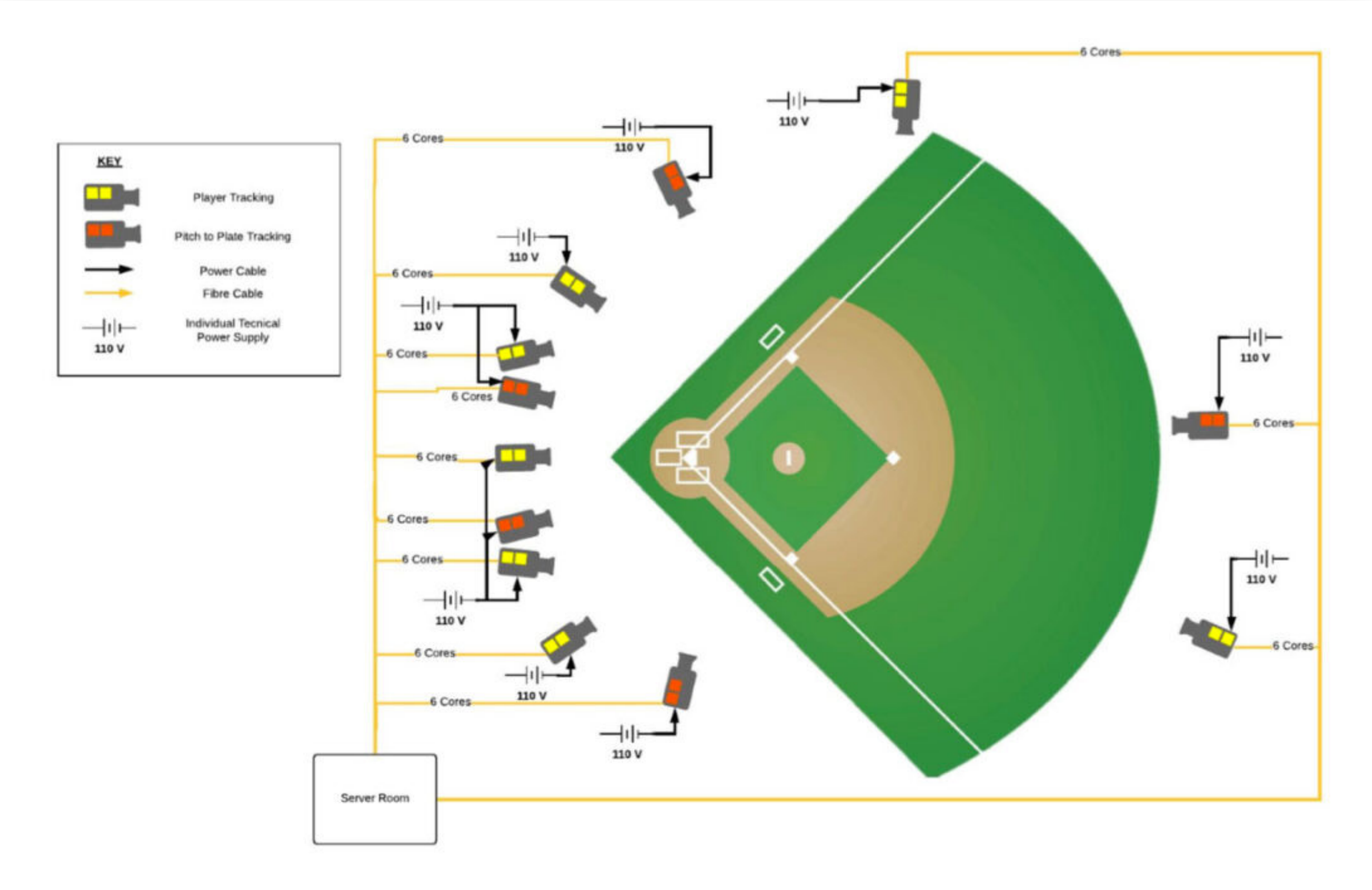


Figure 1: Position and scope of Hawkeye cameras at a baseball stadium

In 2015, Major League Baseball (MLB) introduced Statcast, a set of cameras and radar systems installed in all 30 MLB stadiums. Statcast generates up to seven terabytes of data during a game, capturing every imaginable data point and metric related to pitching, hitting, running and fielding, which the system collects and organizes for consumption. This explosion of data has created opportunities to analyze the game in real time, and with the application of machine learning, teams are now able to make decisions that influence the outcome of the game, pitch by pitch. It's been 20 seasons since the A's first introduced the use of data modeling to baseball. Here's an inside look at how professional baseball teams use technologies like Databricks to create the modern-day "Moneyball" and gain competitive advantages that data teams provide to coaches and players on the field.

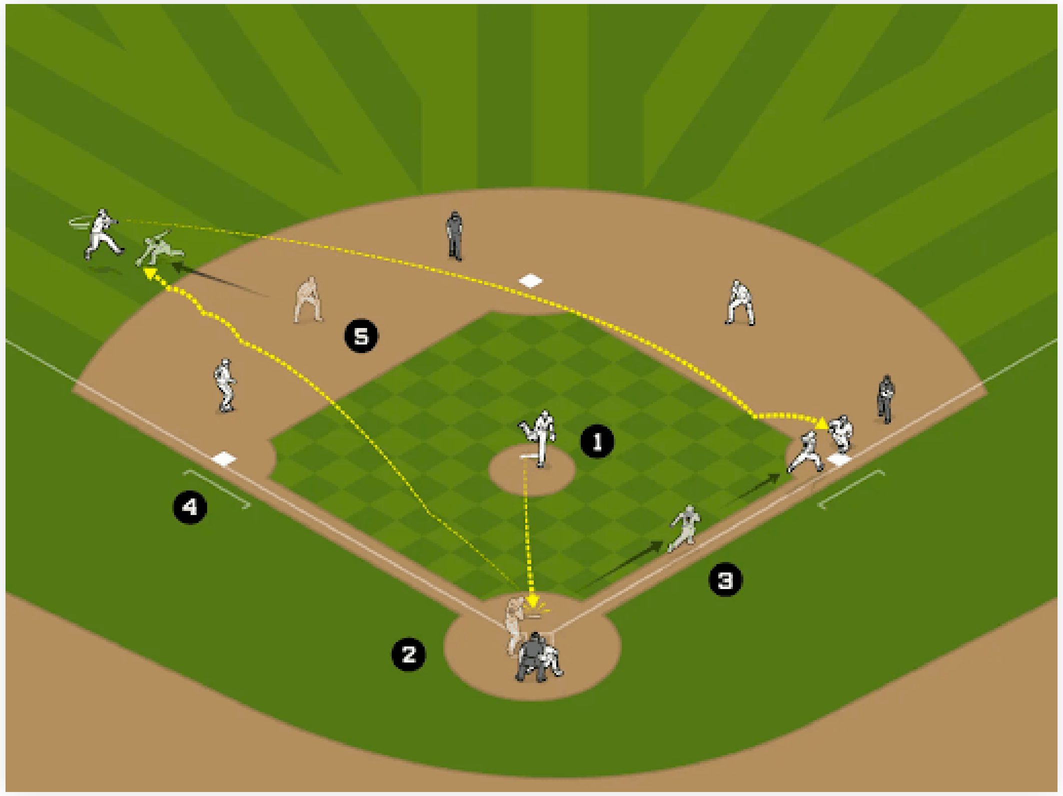
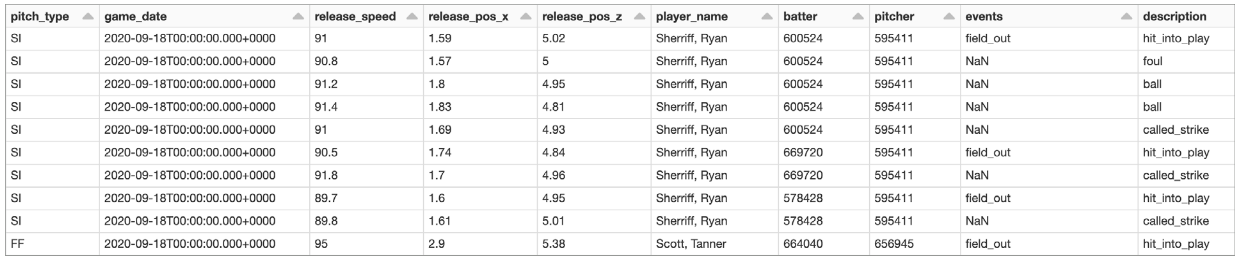


Figure 2: Numbers represent events during a play captured by Statcast



pitch_type	game_date	release_speed	release_pos_x	release_pos_z	player_name	batter	pitcher	events	description
SI	2020-09-18T00:00:00.000+0000	91	1.59	5.02	Sherriff, Ryan	600524	595411	field_out	hit_into_play
SI	2020-09-18T00:00:00.000+0000	90.8	1.57	5	Sherriff, Ryan	600524	595411	NaN	foul
SI	2020-09-18T00:00:00.000+0000	91.2	1.8	4.95	Sherriff, Ryan	600524	595411	NaN	ball
SI	2020-09-18T00:00:00.000+0000	91.4	1.83	4.81	Sherriff, Ryan	600524	595411	NaN	ball
SI	2020-09-18T00:00:00.000+0000	91	1.69	4.93	Sherriff, Ryan	600524	595411	NaN	called_strike
SI	2020-09-18T00:00:00.000+0000	90.5	1.74	4.84	Sherriff, Ryan	669720	595411	field_out	hit_into_play
SI	2020-09-18T00:00:00.000+0000	91.8	1.7	4.96	Sherriff, Ryan	669720	595411	NaN	called_strike
SI	2020-09-18T00:00:00.000+0000	89.7	1.6	4.95	Sherriff, Ryan	578428	595411	field_out	hit_into_play
SI	2020-09-18T00:00:00.000+0000	89.8	1.61	5.01	Sherriff, Ryan	578428	595411	NaN	called_strike
FF	2020-09-18T00:00:00.000+0000	95	2.9	5.38	Scott, Tanner	664040	656945	field_out	hit_into_play

Figure 3: Sample of data collected by Statcast

Background

Data teams need to be faster than ever to provide analytics to coaches and players so they can make decisions as the game unfolds. The decisions made from real-time analytics can dramatically change the outcome of a game and a team's season. One of the more memorable examples of this was in game six of the 2020 World Series. The Tampa Bay Rays were leading the Los Angeles Dodgers 1-0 in the sixth inning when Rays pitcher Blake Snell was pulled from the mound while pitching arguably one of the best games of his career, a decision head coach Kevin Cash said was made with the insights from their data analytics. The Rays went on to lose the game and World Series. Hindsight is always 20-20, but it goes to show how impactful data has become to the game. Coaching staff task their data teams with assisting them in making critical decisions — for example, should a pitcher throw another inning or make a substitution to avoid a potential injury? Does a player have a greater probability of success stealing from first to second base, or from second to third?

I have had the opportunity to work with many MLB franchises and discuss what their priorities and challenges are related to data analytics. Typically, I hear three recurring themes their data teams are focused on that have the most value in helping set their team up for success on the field:

1. **Speed:** Since every MLB team has access to the Statcast data during a game, one way to create a competitive advantage is to ingest and process the data faster than your opponent. The average length of time between pitches is 23 seconds, and this window of time represents a benchmark from which Statcast data can be ingested and processed for coaches to use to make decisions that can impact the outcome of the game.

2. **Real-Time Analytics:** Another competitive advantage for teams is the creation of insights from their machine learning models in real time. An example of this is knowing when to substitute out a pitcher from fatigue, where a model interprets pitcher movement and data points created from the pitch itself and is able to forecast deterioration of performance pitch by pitch.

3. **Ease of Use:** Analytics teams run into problems ingesting the volumes of data Statcast produces when running data pipelines on their local computers. This gets even more complicated when trying to scale their pipelines to capture minor league data and integrate with other technologies. Teams want a collaborative, scalable analytics platform that automates data ingestion with performance, creating the ability to impact in-game decision-making.

Baseball teams using Databricks have developed solutions for these priorities and several others. They have shaped what the modern-day version of "Moneyball" looks like. What follows is their successful framework explained in an easy-to-understand way.

Getting the data

When a pitcher throws a baseball, Hawkeye cameras collect the data and save it to an application that teams are able to access using an application programming interface (API) owned by MLB. You can think of an API as an intermediate connection between two computers to exchange information. The way this works is: a user sends a request to an API, the API confirms that the user has permission to access the data and then sends back the requested data for the user to consume. To use a restaurant as an analogy – a customer tells a waiter what they want to eat, the waiter informs the kitchen what the customer wants to eat, the waiter serves the food to the customer. The waiter in this scenario is the API.

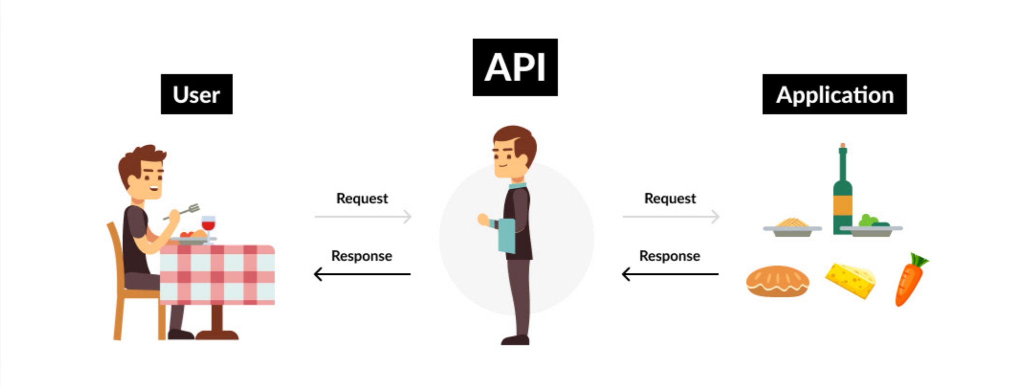


Figure 4: Example of how an API works, using a restaurant analogy

This simple method of retrieving data is called a "batch" style of data collection and processing, where data is gathered and processed once. As noted earlier, however, data is typically available through the API every 23 seconds (the average time between pitches). This means data teams need to make continuous requests to the API in a method known as "streaming," where data is continuously collected and processed. Just as a waiter can quickly become overworked fulfilling customers' needs, making continuous API requests for data creates some challenges in data pipelines. With the assistance from these data teams, however, we have created code to accommodate continuously collecting Statcast data during a game. You can see an example of the code using a test API below.

```
from pathlib import Path
import json

class sports_api:
    def _init_(self, endpoint, api_key):
        self.endpoint = endpoint
        self.api_key = api_key
        self.connection = self.endpoint + self.api_key

    def fetch_payload(self, request_1, request_2, adls_path):
        url = f”{self.connection}&series_id={request_1}{request_2}-
99.M”
        r = requests.get(url)
        json_data = r.json()
        now = time.strftime(“%Y%m%d-%H%M%S”)
        file_name = f”json_data_out_{now}”
        file_path = Path(“dbfs:/”) / Path(adls_path) / Path(file_name)
        dbutils.fs.put(str(file_path), json.dumps(json_data), True)
        return str(file_path)
```

Figure 5: Interacting with an API to retrieve and save data

This code decouples the steps of getting data from the API and transforming it into usable information, which in the past, we have seen, can cause latency in data pipelines. Using this code, the Statcast data is saved as a file to cloud storage automatically and efficiently. The next step is to ingest it for processing.

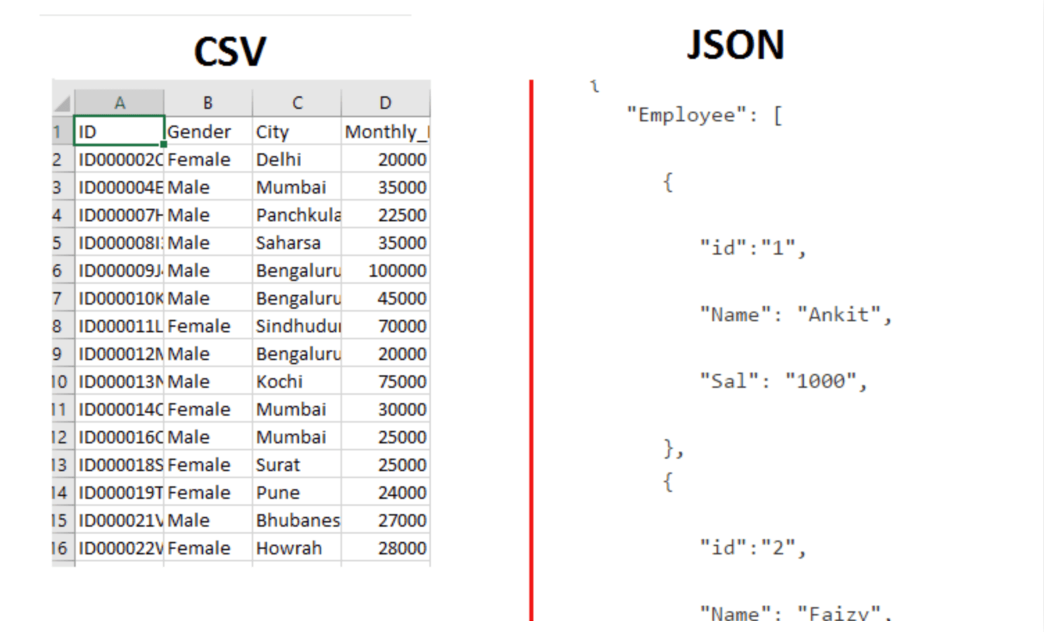
Automatically load data with Auto Loader

As pitch and play data is continuously saved to cloud storage, it can be ingested automatically using a Databricks feature called Auto Loader. Auto Loader scans files in the location they are saved in cloud storage and loads the data into Databricks where data teams begin to transform it for their analytics. Auto Loader is easy to use and incredibly reliable when scaling to ingest larger volumes of data in batch and streaming scenarios. In other words, Auto Loader works just as well for small and large data sizes in batch and streaming scenarios. The Python code below shows how to use Auto Loader for streaming data.

```
df = spark.readStream.format(“cloudFiles”) \
    .option(,) \
    .schema() \
    .load()

df.writeStream.format(“delta”) \
    .option(“checkpointLocation”, ) \
    .trigger() \
    .start()
```

Figure 6: Setup of Auto Loader to stream data

One challenge in this process is working with the file format in which the Statcast is saved, a format called JSON. We are typically privileged to work with data that is already in a structured format, such as the CSV file type, where data is organized in columns and rows. The JSON format organizes data into arrays and despite its wide use and adoption, I still find it difficult to work with, especially in large sizes. Here’s a comparison of data saved in a CSV format and a JSON format.



CSV

	A	B	C	D
1	ID	Gender	City	Monthly_I
2	ID000002C	Female	Delhi	20000
3	ID000004E	Male	Mumbai	35000
4	ID000007H	Male	Panchkula	22500
5	ID000008I:	Male	Saharsa	35000
6	ID000009J.	Male	Bengaluru	100000
7	ID000010K	Male	Bengaluru	45000
8	ID000011L	Female	Sindhudu	70000
9	ID000012N	Male	Bengaluru	20000
10	ID000013N	Male	Kochi	75000
11	ID000014C	Female	Mumbai	30000
12	ID000016C	Male	Mumbai	25000
13	ID000018S	Female	Surat	25000
14	ID000019T	Female	Pune	24000
15	ID000021V	Male	Bhubanes	27000
16	ID000022V	Female	Howrah	28000

JSON

```
{
   "Employee": [

      {

         "id":"1",

         "Name": "Ankit",

         "Sal": "1000",

      },
      {

         "id":"2",

         "Name": "Faizy".
```

Figure 7: Comparison of CSV and JSON formats

It should be obvious which of these two formats data teams prefer to work with. The goal then is to load Statcast data in the JSON format and transform it into the friendlier CSV format. To do this, we can use the semi-structured data support available in Databricks, where basic syntax allows us to extract and transform the nested data you see in the JSON format to the structured CSV style format. Combining the functionality of Auto Loader and the simplicity of semi-structured data support creates a powerful data ingestion method that makes the transformation of JSON data easy.

Using Databricks’ semi-structured data support with Auto Loader

```
spark.readStream.format(“cloudFiles”) \
    .option(“cloudFiles.format”, “json”) \
    .option(“cloudFiles.schemaLocation”, “”) \
    .load(“”) \
    .selectExpr(
    “*”,
    “tags:page.name”, # extracts {“tags”:{“page”:{“name”:...}}}
    “tags:page.id::int”, # extracts {“tags”:{“page”:{“id”:...}}} and
casts to int
    “tags:eventType” # extracts {“tags”:{“eventType”:...}}
    )
```

As the data is loaded in, we save it to a Delta table to start working with it further. Delta Lake is an open format storage layer that brings reliability, security and performance to a data lake for both streaming and batch processing and is the foundation of a cost-effective, highly scalable data platform. Semi-structured support with Delta allows you to retain some of the nested data if needed. The syntax allows flexibility to maintain nested data objects as a column within a Delta table without the need to flatten out all of the JSON data. Baseball analytics teams use Delta to version Statcast data and enforce specific needs to run their analytics on while organizing it in a friendly structured format.

Auto Loader writing data to a Delta table as a stream

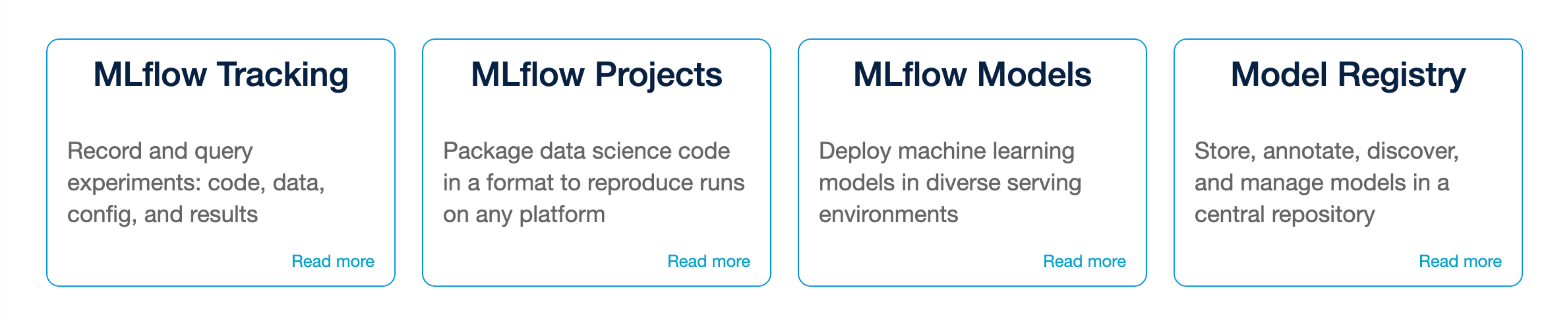
```
# Define the schema and the input, checkpoint, and output paths.
read_schema = (“id int, ” +
    “firstName string, ” +
    “middleName string, ” +
    “lastName string, ” +
    “gender string, ” +
    “birthDate timestamp, ” +
    “ssn string, ” +
    “salary int”)
json_read_path = ‘/FileStore/streaming-uploads/people-10m’
checkpoint_path = ‘/mnt/delta/people-10m/checkpoints’
save_path = ‘/mnt/delta/people-10m’

people_stream = (spark \
    .readStream \
    .schema(read_schema) \
    .option(‘maxFilesPerTrigger’, 1) \
    .option(‘multiline’, True) \
    .json(json_read_path))

people_stream.writeStream \
    .format(‘delta’) \
    .outputMode(‘append’) \
    .option(‘checkpointLocation’, checkpoint_path) \
    .start(save_path)
```

With Auto Loader continuously streaming in data after each pitch, semi-structured data support transforming it into a consumable format, and Delta Lake organizing it for use, data teams are now ready to build analytics that gives their team the competitive edge on the field.

Machine learning for insights

Recall the Rays pulling Blake Snell from the mound during the World Series — that decision came from insights coaches saw in their predictive models. Statistical analysis of Snell's historical Statcast data provided by Billy Heylen of sportingnews.com indicated Snell had not pitched more than six innings since July 2019, had a lower probability of striking out a batter when facing them for the third time in a game, and was being relieved by teammate Nick Anderson, whose own pitch data suggests was one the strongest closers in MLB, with a 0.55 earned run average (ERA) and 0.49 walks and hits per innings pitched (WHIP) during the 19 regular-season games he pitched in 2020. Predictive models analyze data like this in real time and provide supporting evidence and recommendations coaches use to make critical decisions.

Machine learning models are relatively easy to build and use, but data teams often struggle to implement them into streaming use cases. Add in the complexity of how models are managed and stored and machine learning can quickly become out of reach. Fortunately, data teams use MLflow to manage their machine learning models and implement them into their data pipelines. MLflow is an open source platform for managing the end-to-end machine learning lifecycle and includes support for tracking predictive results, a model registry for centralizing models that are in use and others in development, and a serving capability for using models in data pipelines.



MLflow Tracking

Record and query experiments: code, data, config, and results

Read more

MLflow Projects

Package data science code in a format to reproduce runs on any platform

Read more

MLflow Models

Deploy machine learning models in diverse serving environments

Read more

Model Registry

Store, annotate, discover, and manage models in a central repository

Read more

Figure 8: MLflow overview

To implement machine learning algorithms and models to real-time use cases, data teams use the model registry where a model is able to read data sitting in a Delta table and create predictions that are then used during the game. Here’s an example of how to use a machine learning model while data is automatically loaded with Auto Loader:

Getting a machine learning model from the registry and using it with Auto Loader

```
#get model from the model registry
model = mlflow.spark.load_model(
    model_uri=f”models:/{model_name}/{‘Production’}”)

#read data from bronze table as a stream
events = spark.readStream \
    .format(“delta”) \
    #.option(“cloudFiles.maxFilesPerTrigger”, 1)\
    .schema(schema) \
    .table(“baseball_stream_bronze”)

#pass stream through model
model_output = model.transform(events)

#write stream to silver delta table
events.writeStream \
    .format(‘delta’) \
    .outputMode(“append”) \
    .option(‘checkpointLocation’, “/tmp/baseball/”) \
    .table(“default.baseball_stream_silver”)
```

The outputs a machine learning model creates can then be displayed in a data visualization or dashboard and used as printouts or shared on a tablet during a game. MLB franchises working on Databricks are developing fascinating use cases that are being used during games throughout the season. Predictive models are proprietary to the individual teams, but here’s an actual use case running on Databricks that demonstrates the power of real-time analytics in baseball.

Bringing it all together with spin ratios and sticky stuff

MLB introduced a new rule for the 2021 season meant to discourage pitcher's use of "sticky stuff," a substance hidden in mitts, belts or hats that when applied to a baseball can dramatically increase the spin ratio of a pitch, making it difficult for batters to hit. The rule suspends for 10 games pitchers discovered using sticky stuff. Coaches on opposing teams have the ability to request an umpire check for the substance if they suspect a pitcher to be using it during a game. Spin ratio is a data point that is captured by Hawkeye cameras, and with real-time analytics and machine learning, teams are now able to make justified requests to umpires with the hopes of catching a pitcher using the material.

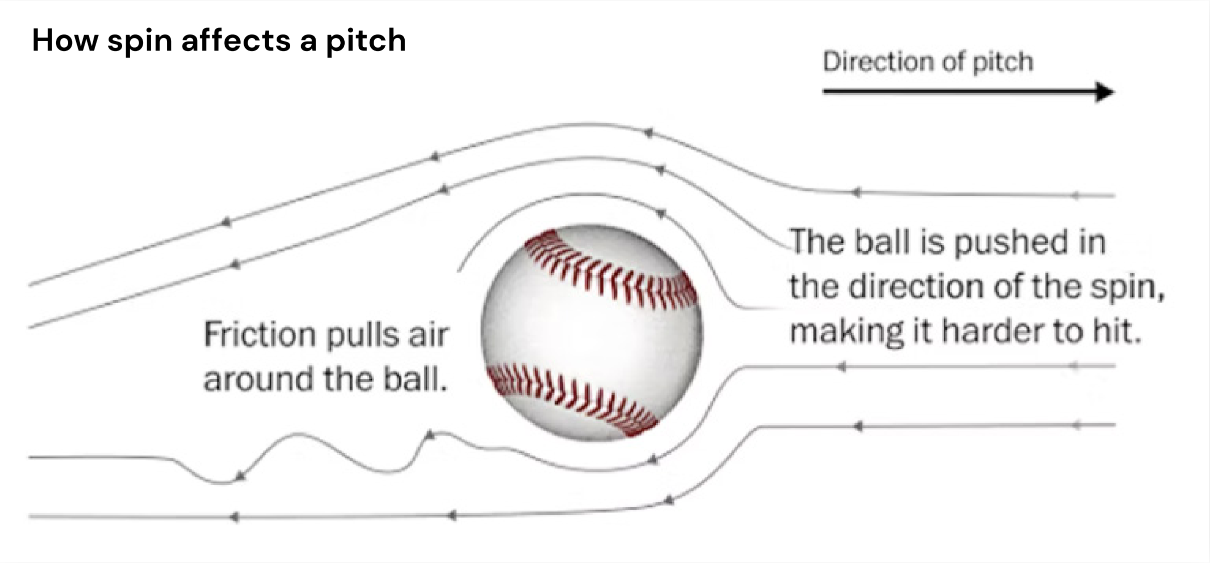

How spin affects a pitch
Direction of pitch
The ball is pushed in the direction of the spin, making it harder to hit.
Friction pulls air around the ball.

Figure 9: Illustration of how spin affects a pitch

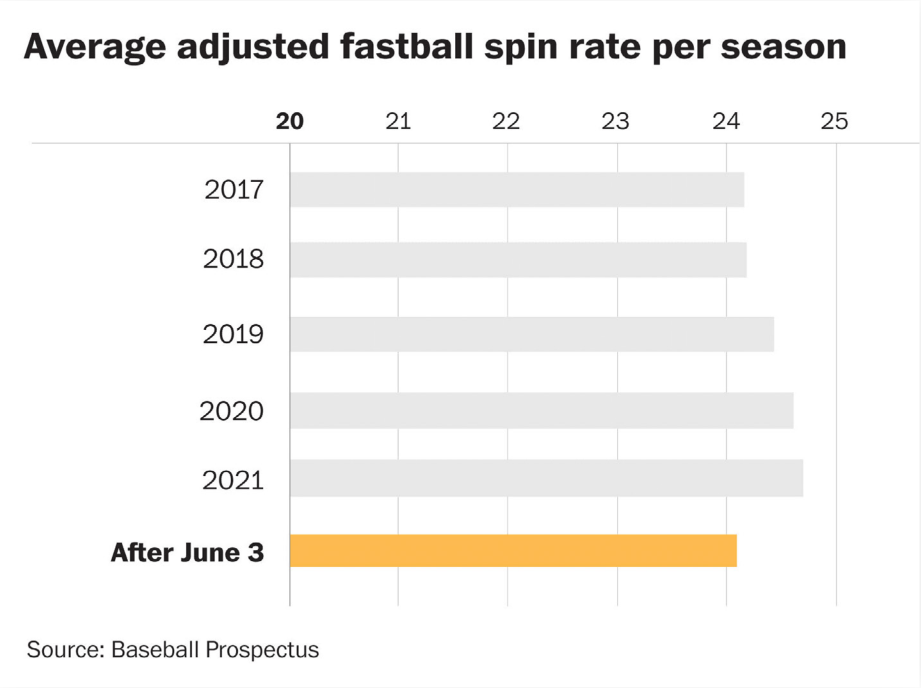
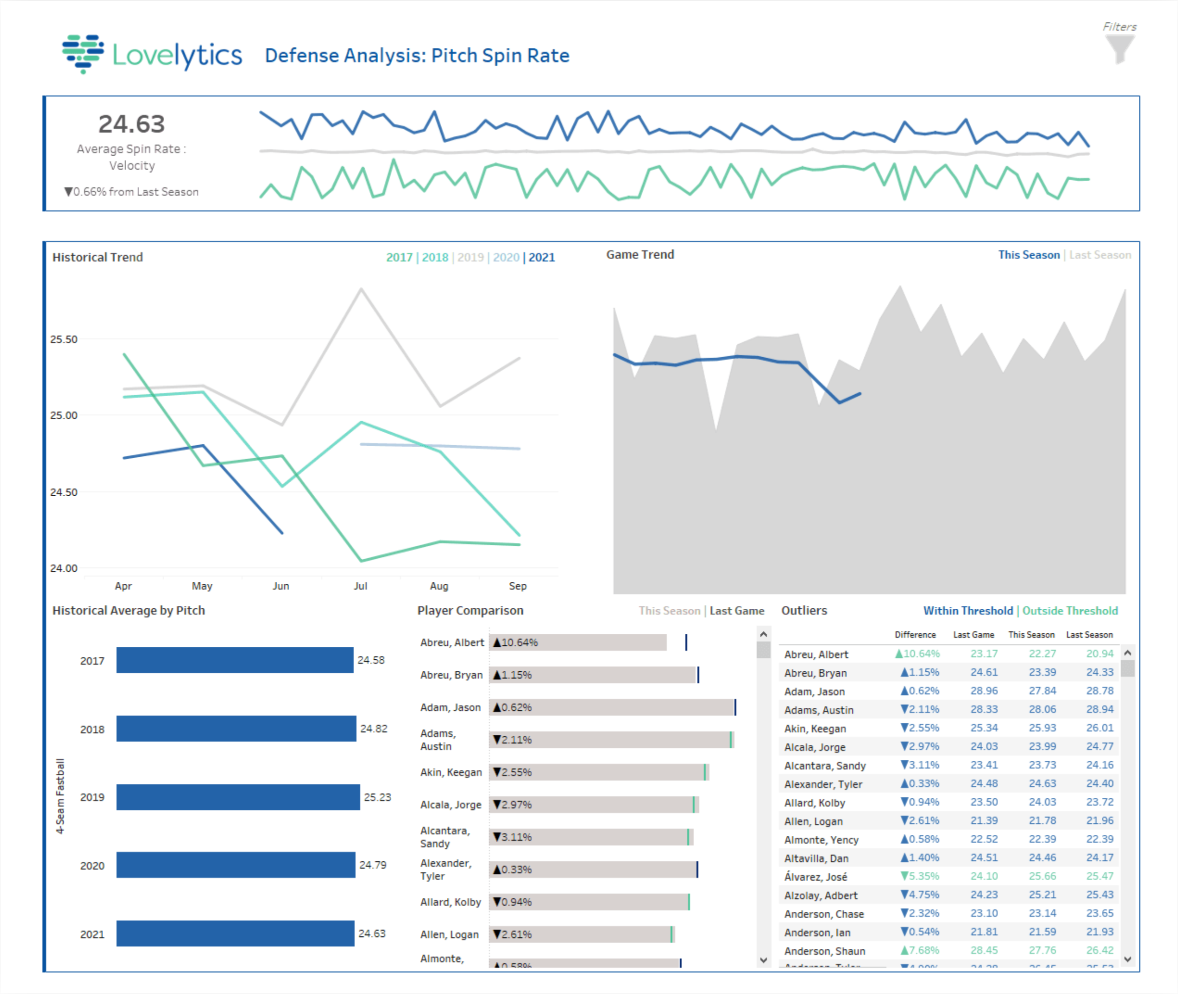


Figure 10: Trending spin rate of fastballs per season and after rule introduction on June 3, 2021

Following the same framework outlined above, we ingest Statcast data pitch by pitch and have a dashboard that tracks the spin ratio of the ball for all pitchers during all MLB games. Using machine learning models, predictions are sent to the dashboard that flag outliers against historical data and the pitcher's performance in the active game, which can alert coaches when they fall outside of ranges anticipated by the model. With Auto Loader, Delta Lake and MLflow, all data ingestion and analytics happen in real time.

Technologies like Statcast and Databricks have brought real-time analytics to sports and changed the paradigm of what it means to be a data-driven team. As data volumes continue to grow, having the right architecture in place to capture real-time insights will be critical to staying one step ahead of the competition. Real-time architectures will be increasingly important as teams acquire and develop players, plan for the season and develop an analytically enhanced approach to their franchise. Ask about our Solution Accelerator with Databricks partner Lovelytics, which provides sports teams with all the resources they need to quickly create use cases like the ones described in this blog.



	Difference	Last Game	This Season	Last Season
Abreu, Albert	▲10.64%	23.17	22.27	20.94
Abreu, Bryan	▲1.15%	24.61	23.39	24.33
Adam, Jason	▲0.62%	28.96	27.84	28.78
Adams, Austin	▼2.11%	28.33	28.06	28.94
Akin, Keegan	▼2.55%	25.34	25.93	26.01
Alcala, Jorge	▼2.97%	24.03	23.99	24.77
Alcantara, Sandy	▼3.11%	23.41	23.73	24.16
Alexander, Tyler	▲0.33%	24.48	24.63	24.40
Allard, Kolby	▼0.94%	23.50	24.03	23.72
Allen, Logan	▼2.61%	21.39	21.78	21.96
Almonte, Yency	▲0.58%	22.52	22.39	22.39
Altavilla, Dan	▲1.40%	24.51	24.46	24.17
Álvarez, José	▼5.35%	24.10	25.66	25.47
Alzolay, Adbert	▼4.75%	24.23	25.21	25.43
Anderson, Chase	▼2.32%	23.10	23.14	23.65
Anderson, Ian	▼0.54%	21.81	21.59	21.93
Anderson, Shaun	▲7.68%	28.45	27.76	26.42

Figure 11: Dashboard for “sticky stuff” detection in real time

CHAPTER 3:

Improving On-Shelf Availability for Items With AI Out-of-Stock Modeling

This post was written in collaboration with Databricks partner Tredence. We thank Rich Williams, Vice President Data Engineering, and Morgan Seybert, Chief Business Officer, of Tredence for their contributions.

By **Rich Williams, Morgan Seybert, Rob Saker** and **Bryan Smith**

Introduction

Retailers are missing out on nearly $1 trillion in global sales because they don't have on hand what customers want to buy in their stores. Adding to the challenge, a study of 600 households and several retailers by research firm IHL Group details that shoppers encounter out-of-stocks (OOS) as often as one in three shopping trips, according to the report. And a study by IRI found that 20% of all out-of-stocks remain unresolved for more than 3 days.

Overall, studies show that the average OOS rate is about 8%. That means that one out of 13 products is not purchasable at the exact moment the customer wants to get it in the store. OOS is one of the biggest problems in retail, but thankfully it can be solved with real-time data and analytics.

In this write-up, we showcase the new Tredence-Databricks combined On-Shelf Availability Solution Accelerator. The accelerator is a robust quick-start guide that is the foundation for a full out-of-stock or supply chain solution. We outline how to approach out-of-stocks with the Databricks Lakehouse to solve for on-shelf availability in real time.

And the impact of solving this problem? A 2% improvement in on-shelf availability is worth 1% in increased sales for retailers.

Growth in e-commerce makes item availability more important

The significance of this problem has been amplified by the availability of e-commerce for delivery and curbside pickup orders. While customers that face an out-of-stock at the store level may just not purchase that item, they are likely to purchase other items in the store. Buying online means that they may just switch to a different retailer.

The impact is not just limited to a bottom line loss in revenue. Research from NielsenIQ shows that 30% of shoppers will visit new stores when they can't find the product they are looking for, leading to a loss in long-term loyalty. Members of e-commerce membership programs are most likely to switch retailers in the event of an out-of-stock. IHL estimates that "upwards of 24% of Amazon's current retail revenue comes from customers who first tried to buy the product in-store."

Retailers have responded to this with a variety of tactics including over-ordering of items, which increases carrying costs and lowers margins when they are forced to sell excess inventory at a discount. In some instances, retailers and distributors will rush order products or use intra-delivery "hot shots" for additional deliveries, which come at an additional cost. Some retailers have invested in robotics, but many pull out of their pilots citing costs. And other retailers are experimenting with computer vision, although these approaches merely notify them when an item is unavailable and don't predict item availability.

It's not just retailers that are impacted by OOS. Retailers, consumer goods companies, distributors, brokers and other firms each invest in third-party audits, which typically involve employees visiting stores to identify gaps on the shelf. On any given day, tens of thousands of individuals are visiting stores to validate item availability. Is this really the best use of time and resources?

Why hasn't technology solved out-of-stocks yet?

Out-of-stock issues have been around for decades, so why hasn't the retail industry been able to solve an issue of this magnitude that impacts shoppers, retailers and brands alike? The seemingly simple solution is to require employees to manually count the items on hand. But with potentially hundreds of thousands of individual SKUs distributed across a large format retail location that may be servicing customers nearly 24 hours a day, this simply isn't a realistic task to perform on a regular basis.

Individual stores do perform inventory counts periodically and then rely on point-of-sale (POS) and inventory management software to track changes that drive unit counts up and down. But with so much activity within a store location, some of the day-to-day recordkeeping falls through the cracks, not to mention the impact of shrinkage, which can be hard to detect, on in-store supplies.

So the industry falls back on modeling. But given fundamental problems in data accuracy, these approaches can drive a combination of false positives and false negatives that make model predictions difficult to employ. Time sensitivities further exacerbate the problem, as the large volume of data that often must be crunched in order to arrive at model predictions must be handled fast enough for the results to be actionable. The problem of building a reliable system for stockout prediction and alerting is not as straightforward as it might appear.

Introducing the On-Shelf Availability Solution Accelerator

Our partners at Tredence approached us with the idea of publishing a Solution Accelerator that they've created as the core of a broader Supply Chain Control Tower offering. Tredence works with the largest retailers on the planet and understands the nuances of modeling OOS and knew that Databricks' processing and their advanced data science capabilities were a winning combination.

While the OSA solution focuses on driving sales through improved stock availability on the shelves, the broader Retail Supply Chain Control Tower solves for multiple adjacent merchandising problems – inventory design for the stores, efficient store replenishments, design of store network for omnichannel operations, etc. Knowing how big a problem this is in retail, we immediately took them up on their offer.

The first step in addressing OSA challenges is to examine their occurrence in the historical data. Past occurrences point to systemic issues with suppliers and internal processes, which will continue to cause problems if not addressed.

To support this analysis, Tredence made available a set of historical inventory and sales data. These data sets were simulated, given the obvious sensitivities any retailer would have around this information, but were created in a manner that frequently observed OSA challenges manifested in the data. These challenges were:

1. **Phantom inventory**
2. **Safety stock violations**
3. **Zero-sales events**
4. **On-shelf availability**

Phantom inventory

In a phantom inventory scenario, the units reported to be on hand do not align with units expected based on reported sales and replenishment.

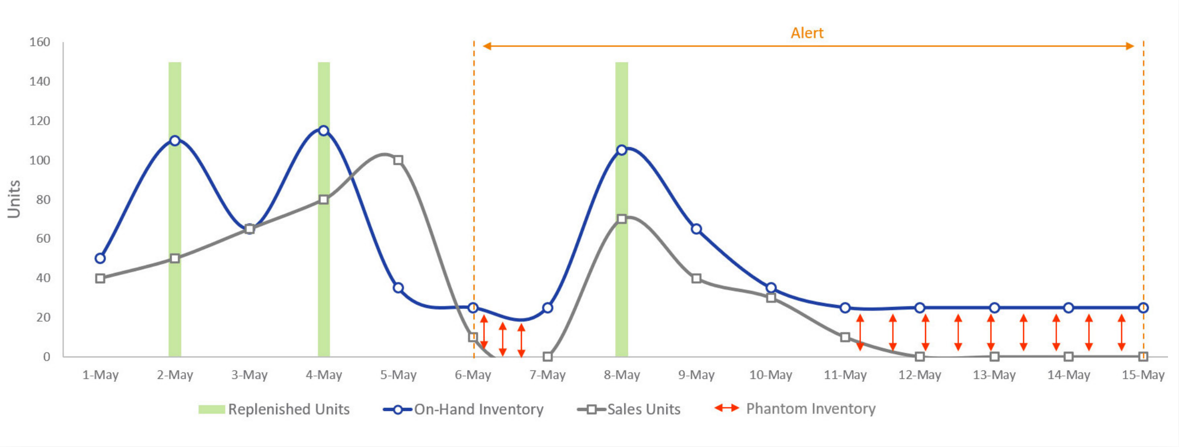


Figure 1: The misalignment of reported inventory, with inventory expected based on sales and replenishment, creating phantom inventory

Poor tracking of replenishment units, unreported or undetected shrinkage, and out-of-band processes coupled with infrequent and sometimes inaccurate inventory counts create a situation where retailers believe they have more units on hand than they actually do. If large enough, this phantom inventory may delay or even prevent the ordering of replenishment units, leading to an out-of-stock scenario.

Safety stock violations

Most organizations establish a threshold for a given product's inventory, below which replenishment orders are triggered. If set too low, inadequate lead times or even minor disruptions to the supply chain may lead to an out-of-stock scenario while new units are moving through the replenishment pipeline.

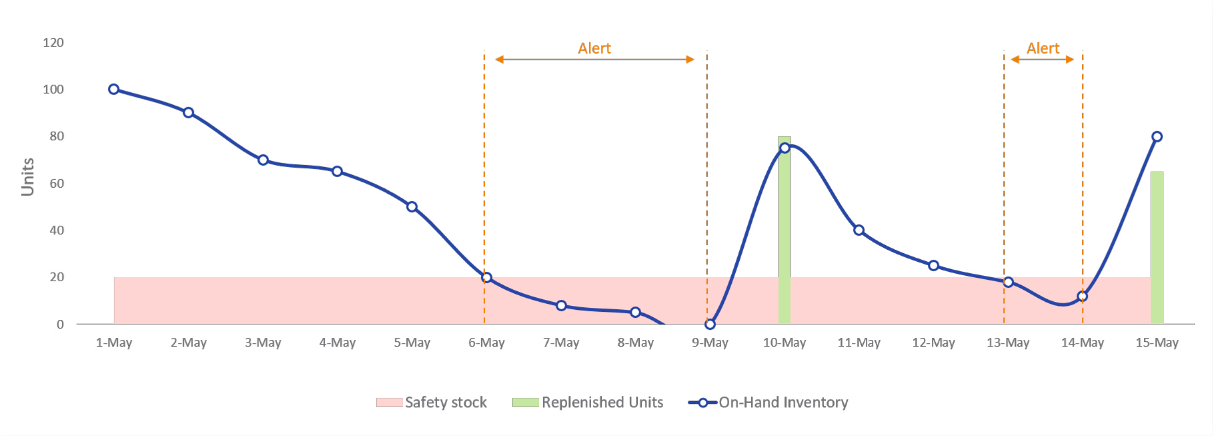


Figure 2: Safety stock levels not providing adequate lead time to prevent out-of-stock issues

The flip side of this is that if set too high, retailers risk overstocking products that may expire, risk damage or theft, or otherwise consume space and capital that may be better employed in other areas. Finding the right safety stock level for a product in a specific location is a critical task for effective inventory management.

Zero-sales events

Phantom inventory and safety stock violations are the two most common causes of out-of-stocks. Regardless of the cause, out-of-stock events manifest themselves in periods when no units of a product are sold.

Not every occurrence of a zero-sales event reflects an out-of-stock concern. Some products don't sell every day, and for some slow-moving products, multiple days may go by within which zero units are sold while the product remains adequately stocked.

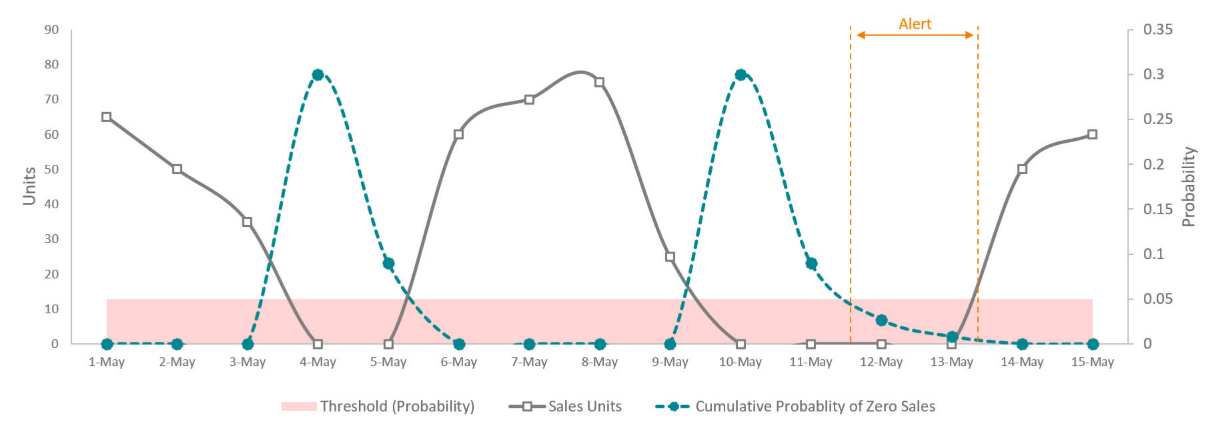


Figure 3: Examining the cumulative probability of consecutive zero-sales events to identify potential out-of-stock issues

The trick for scrutinizing zero-sales events at the item level is to understand the probability of which at least one unit of a product sells on a given day and to then set a cumulative probability threshold for consecutive days reflecting zero-sales. When the cumulative probability of back-to-back zero-sales events exceeds the threshold, it's time for the inventory of that product to be examined.

On-shelf availability

While understanding scenarios in which items are not in stock is critical, it's equally important to recognize when products are technically available for sale but underperforming because of non-optimal inventory management practices. These merchandising problems may be due to poor placement of displays within the store, the stocking of products deep within a shelf, the slow transfer of product from the backroom to shelves, or a myriad of other scenarios in which inventory is adequate to meet demand but customers cannot easily view or access it.

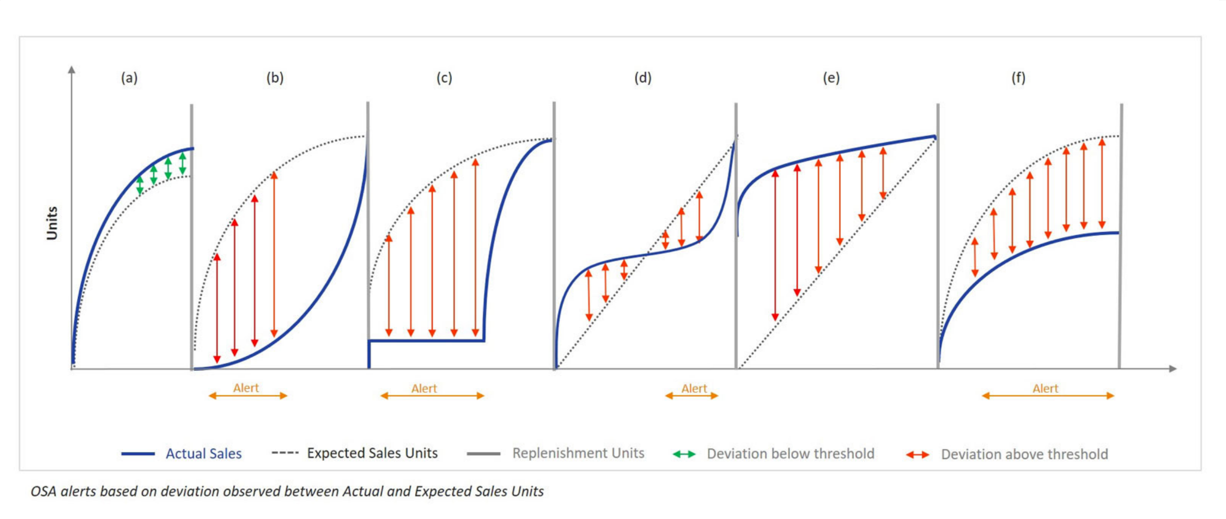


Figure 4: Depressed sales due to poor product placement leading to an on-shelf availability problem

To detect these kinds of problems, it is helpful to compare actual sales to those forecasted for the period. While not every missed sales goal indicates an on-shelf availability problem, a sustained miss might signal a problem that requires further attention.

How we approach out-of-stocks with the Databricks Lakehouse Platform

The evaluation of phantom inventories, safety stock violations, zero-sales events and on-shelf availability problems requires a platform capable of performing a wide range of tasks. Inventory and sales data must be aggregated and reconciled at a per-period level. Complex logic must be applied across these data to examine aggregate and series patterns. Forecasts may need to be generated for a wide range of products across numerous locations. And the results of all this work must be made accessible to the business analysts responsible for scrutinizing the findings before soliciting action from those in the field.

Databricks provides a single platform capable of all this work. The elastic scalability of the platform ensures that the processing of large volumes of data can be performed in an efficient and timely manner. The flexibility of its development environment allows data engineers to pivot between common languages, such as SQL and Python, to perform data analysis in a variety of modes.

Pre-integrated libraries provide support for classic time series forecasting algorithms and techniques, and easy programmatic installations of alternative libraries such as Facebook Prophet allow data scientists to deliver the right forecast for the business's needs. Scalable patterns ensure data science tasks are also tackled in an efficient and timely manner with little deviation from the standard approaches data scientists typically employ.

And the SQL Analytics interface, as well as robust integrations with Tableau and Power BI, allows analysts to consume the results of the data scientists' and data engineers' work without having to first port the data to alternative platforms.

Getting started

Be sure to check out and download the notebooks for out-of-stock modeling. As with any of our Solution Accelerators, these are a foundation for a full solution. If you would like help with implementing a full out-of-stock or supply chain solution, go visit our friends at Tredence.

To see these features in action, please check out the following notebooks demonstrating how Tredence tackled out-of-stocks on the Databricks platform:

OSA 1: Data Preparation →
OSA 2: Out-of-Stocks →
OSA 3: On-Shelf Availability →

CHAPTER 4:

Using Dynamic Time Warping and MLflow to Detect Sales Trends

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

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

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:

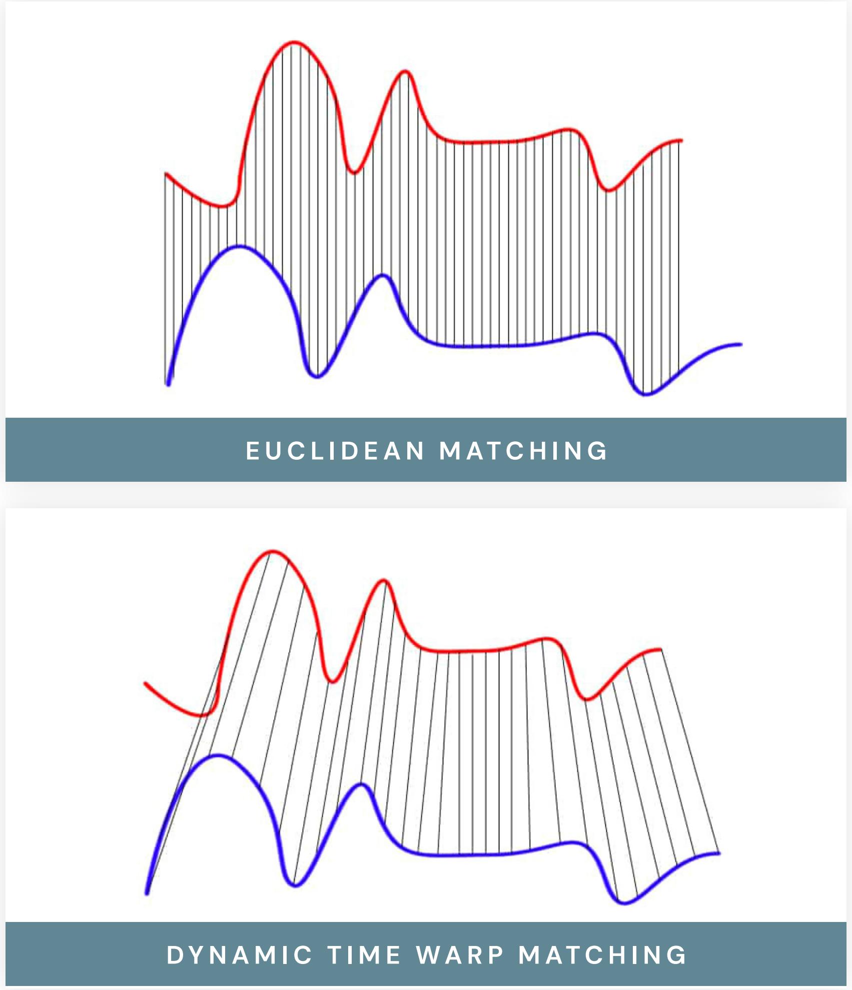
- The basic principles of dynamic time warping
- Running dynamic time warping on sample audio data
- Running dynamic time warping on sample sales data using MLflow

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 disjoint 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.

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

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

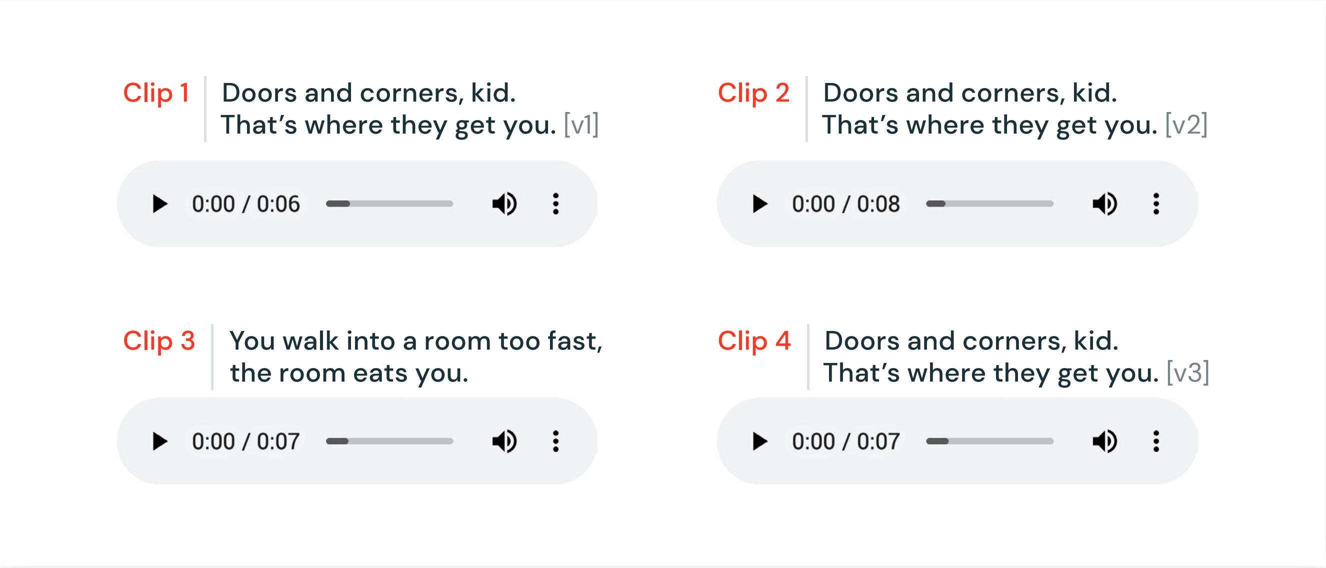
Sound pattern matching

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

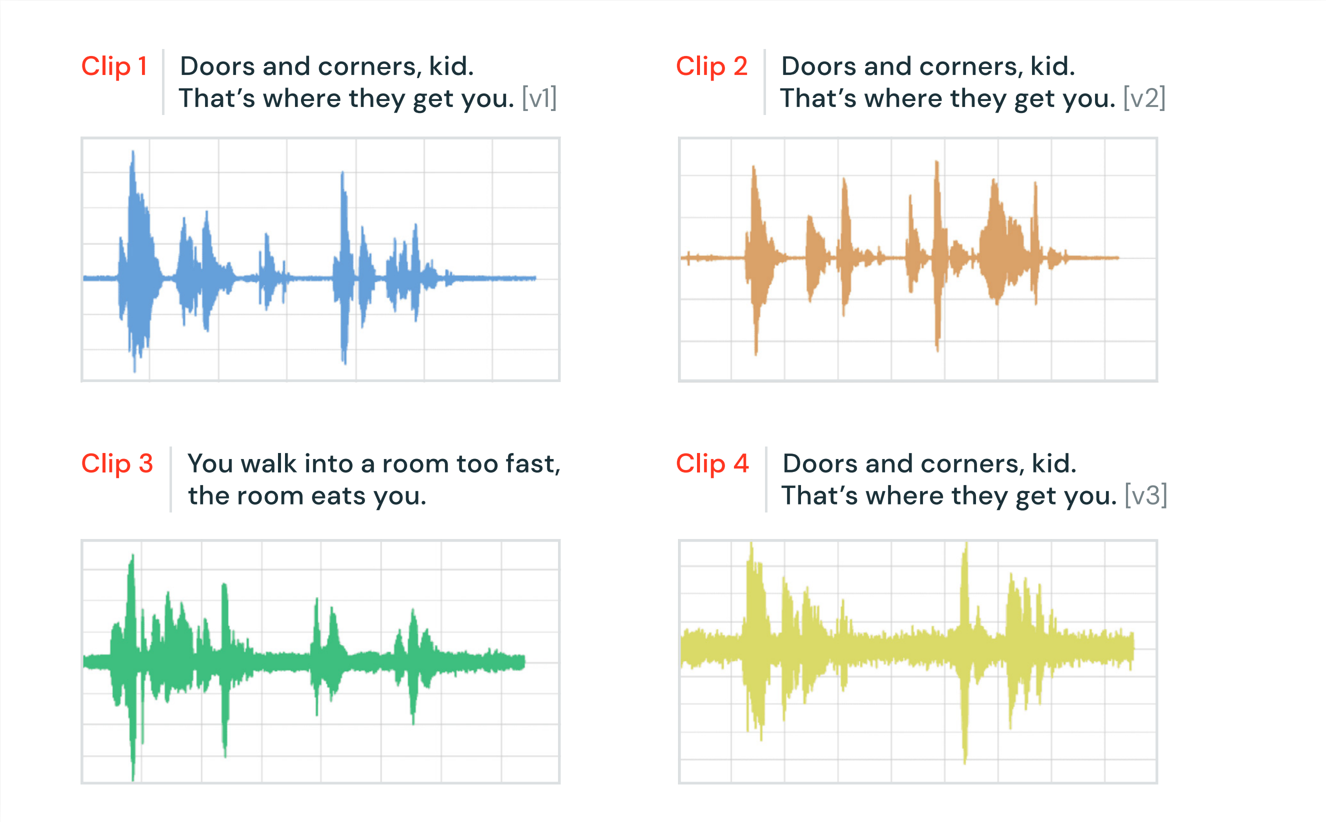
Clip 4 Doors and corners, kid. That's where they get you. [v3]

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.

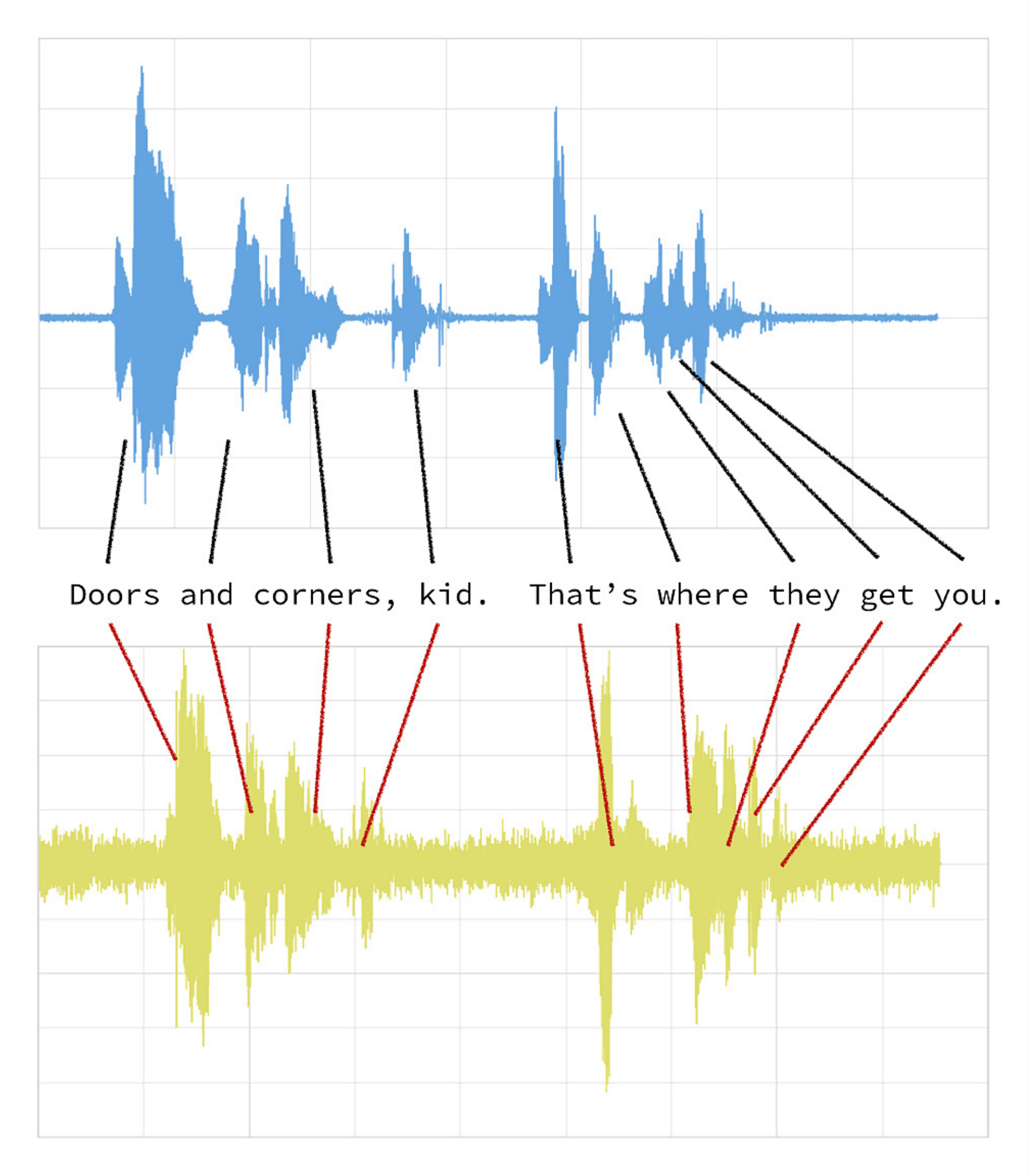
Clip 4 Doors and corners, kid. That's where they get you. [v3]

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

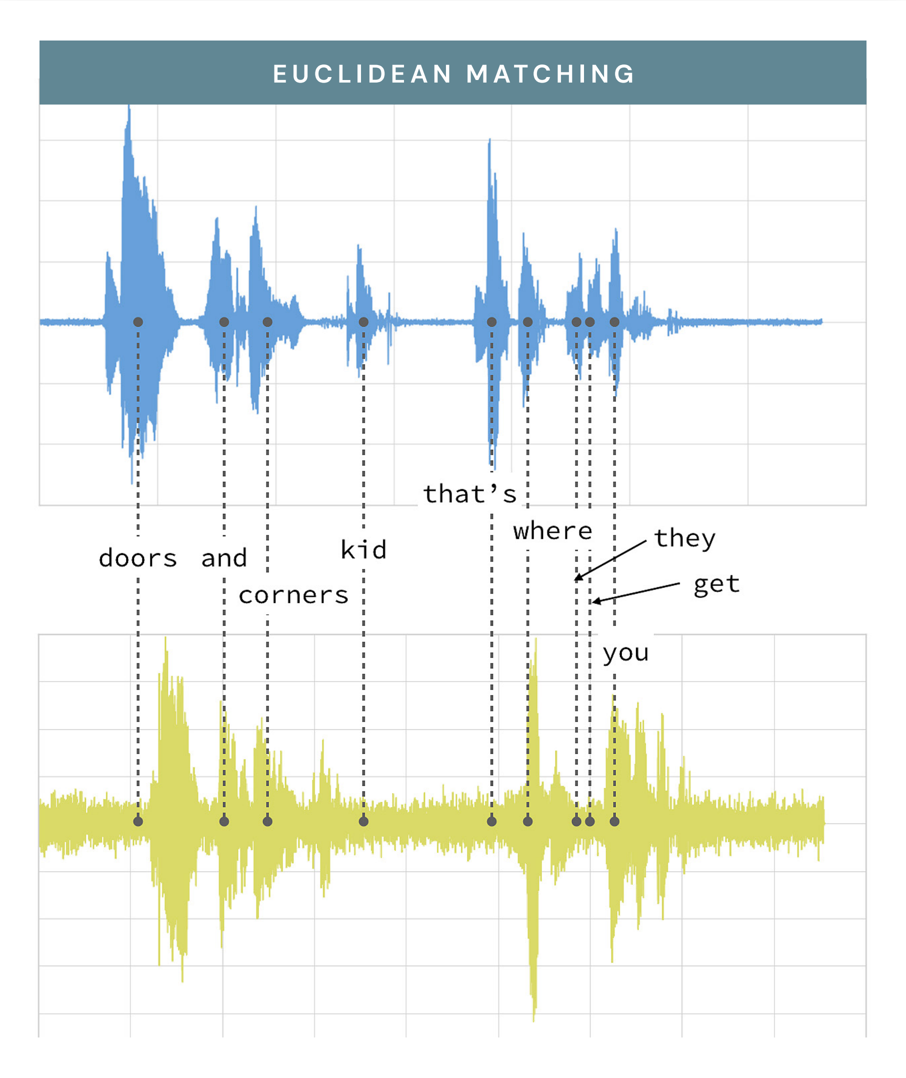
```
from scipy.io import wavfile
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure

# 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.

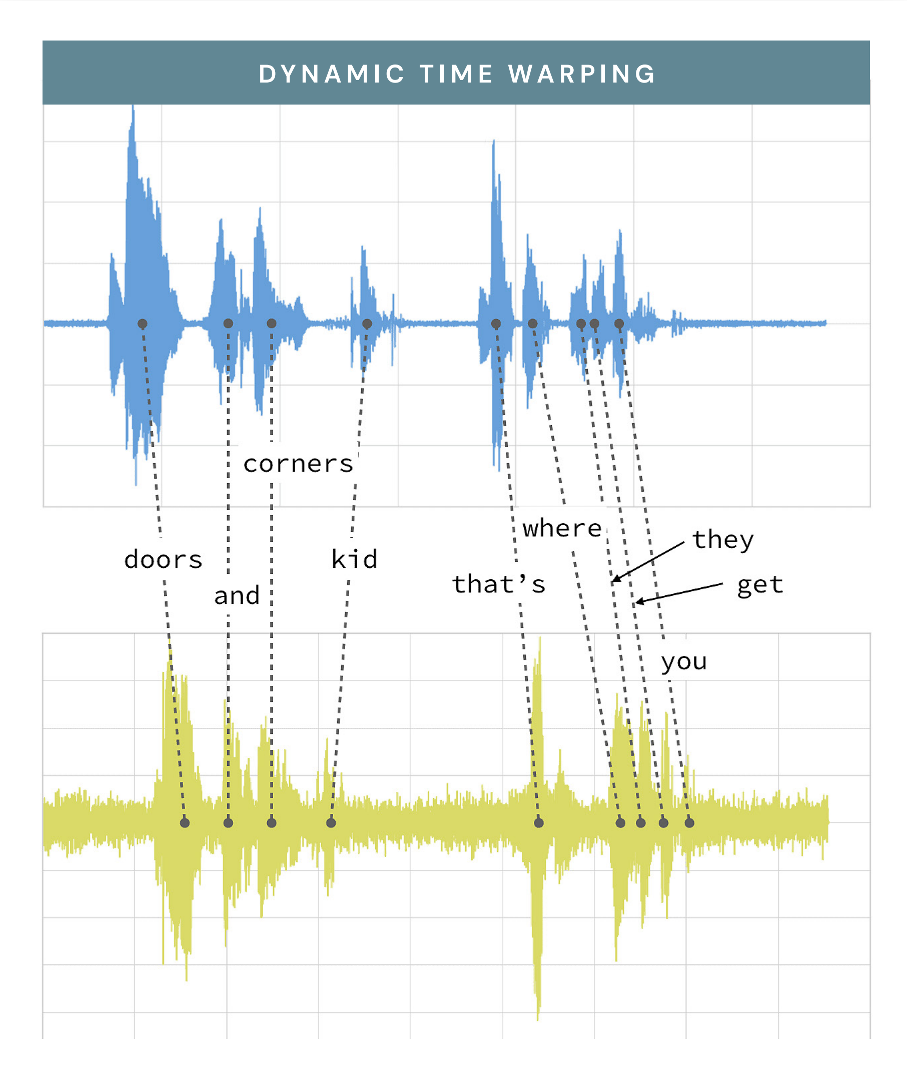
As noted below, the two clips (in this case, clips 1 and 4) have different intonations (amplitude) and latencies for the same quote.



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.

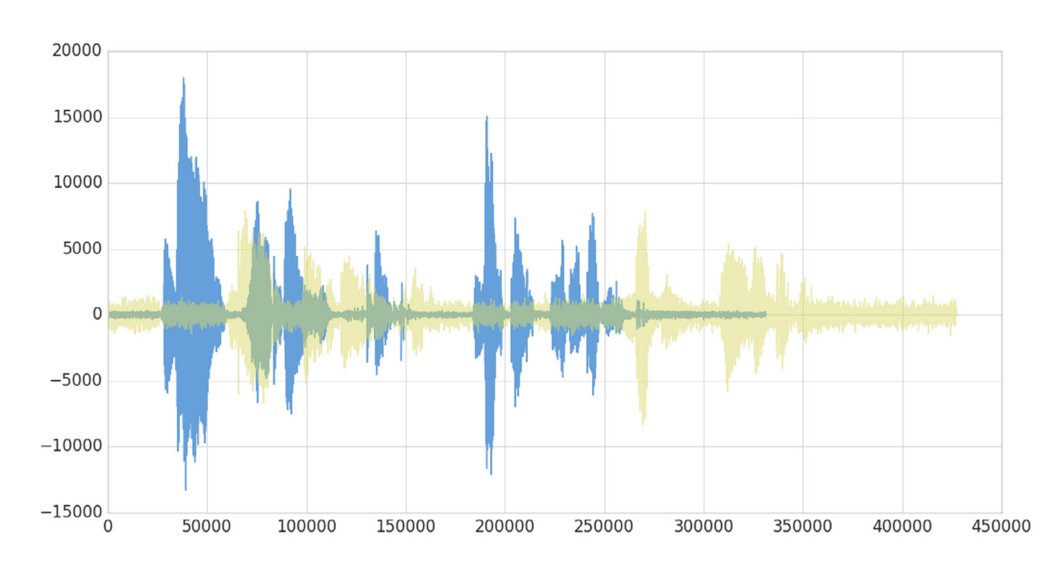


For our time series comparison, we will use the `fastdtw` PyPi library; the instructions to install PyPi libraries within your Databricks workspace can be found here: Azure | AWS. By using fastdtw, we can quickly calculate the distance between the different time series.

```
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 4:

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

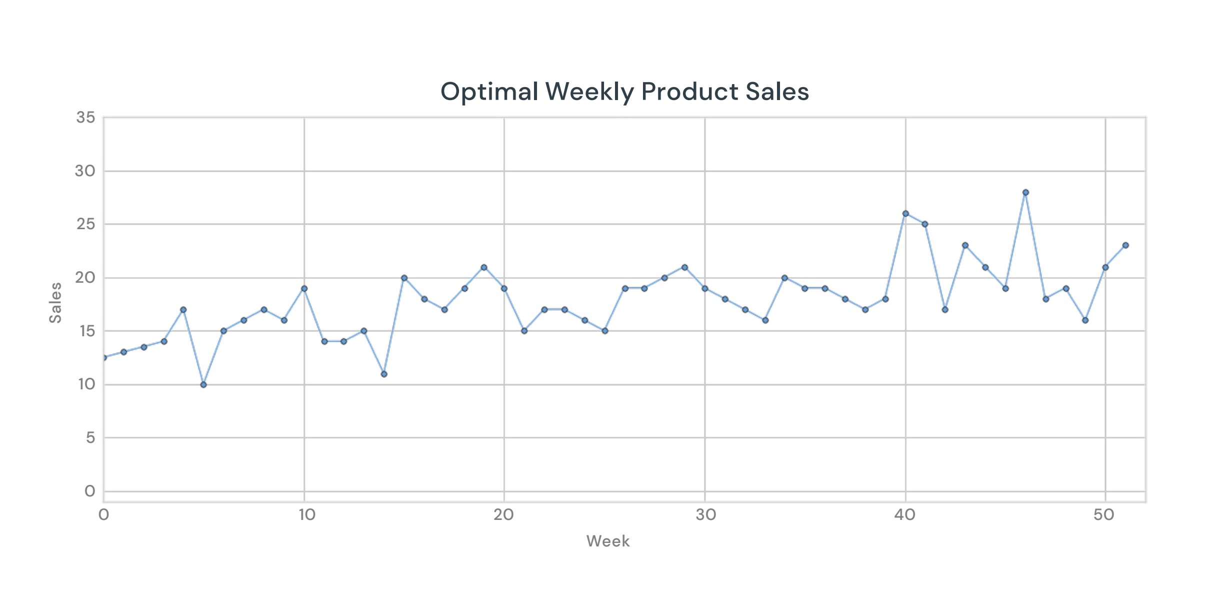
By **Ricardo Portilla**, **Brenner Heintz** and **Denny Lee**

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:



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 which includes last year's sales numbers for each product, and you think this year's sales will be similar.

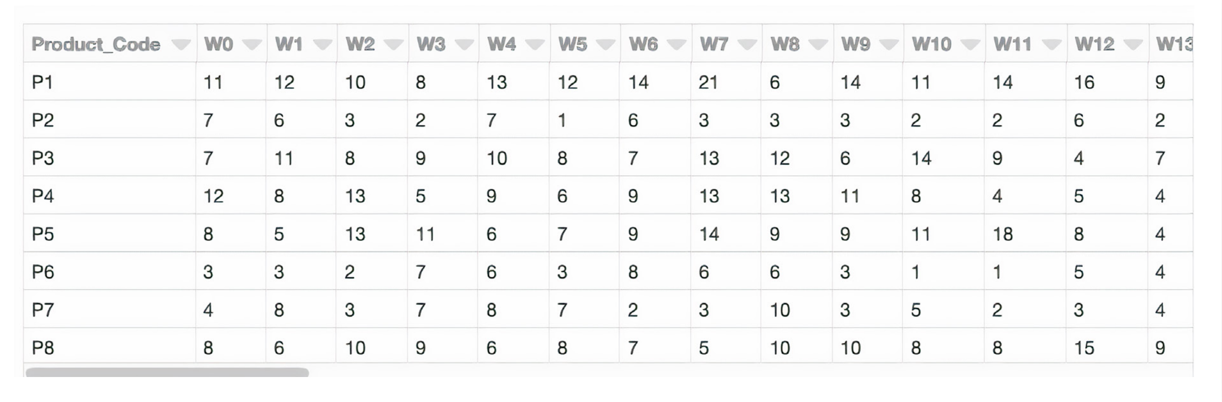
If you choose a product with weekly demand that exceeds your production capacity, then you'll have to cancel customer orders, which isn't good for business. On the other hand, if you choose a product without enough weekly demand, you won't be able to keep your printers running at full capacity and may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



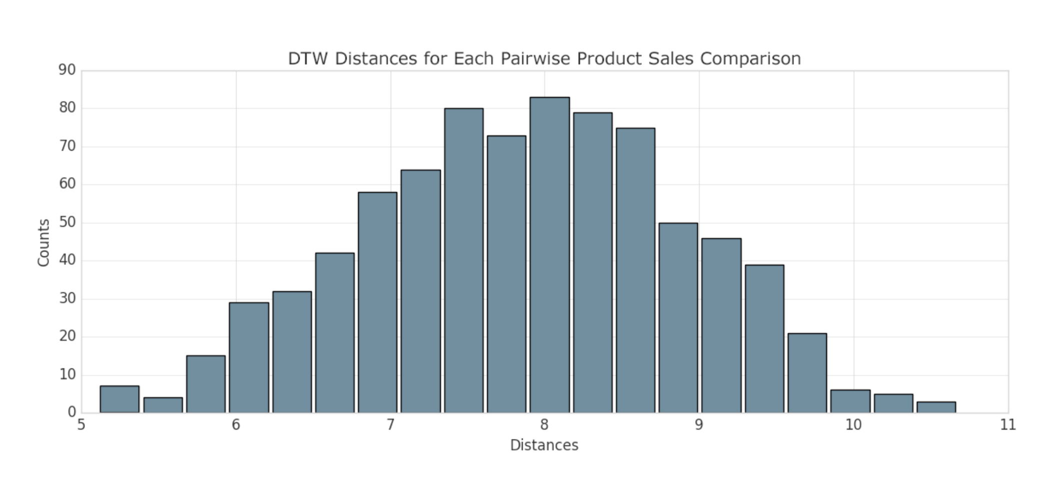
Product_Code	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
P1	11	12	10	8	13	12	14	21	6	14	11	14	16	9
P2	7	6	3	2	7	1	6	3	3	3	2	2	6	2
P3	7	11	8	9	10	8	7	13	12	6	14	9	4	7
P4	12	8	13	5	9	6	9	13	13	11	8	4	5	4
P5	8	5	13	11	6	7	9	14	9	9	11	18	8	4
P6	3	3	2	7	6	3	8	6	6	3	1	1	5	4
P7	4	8	3	7	8	7	2	3	10	3	5	2	3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

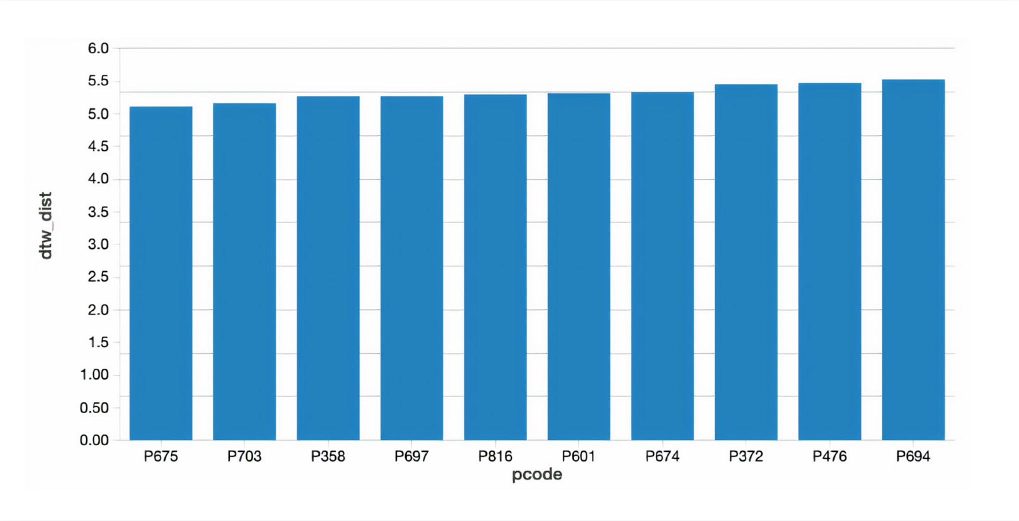
```
# Calculate distance via dynamic time warping between product code and
optimal time series
import numpy as np
import _ucrdtw

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.

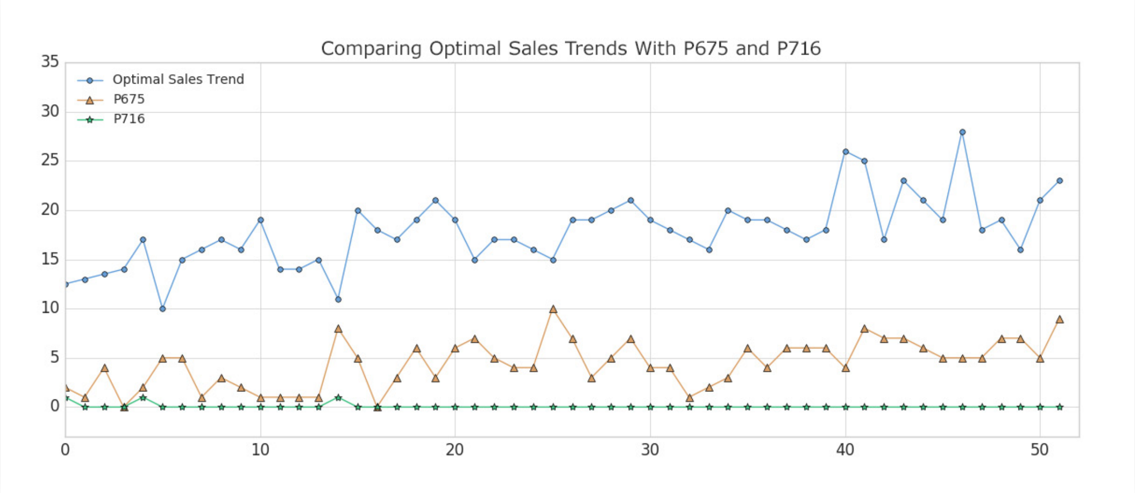


From there, we can identify the product codes closest to the optimal sales trend (i.e., those that have the smallest calculated DTW distance). Since we're using Databricks, we can easily make this selection using a SQL query. Let's display those that are closest.

```
%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.

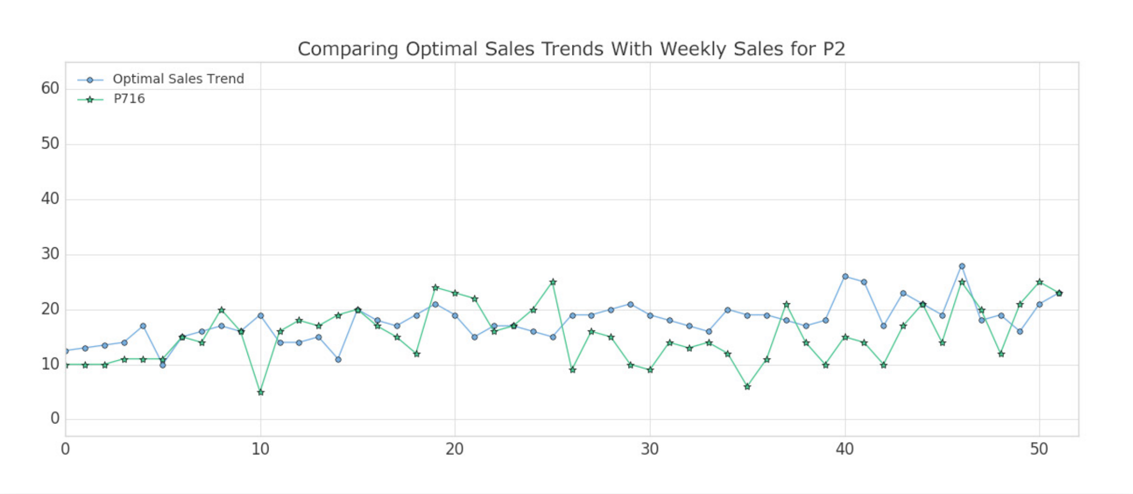


As you can see, Product #675 (shown in the orange triangles) represents the best match to the optimal sales trend, although the absolute weekly sales are lower than we'd like (we'll remedy that later). This result makes sense since we'd expect the product with the closest DTW distance to have peaks and 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
    <strong>with mlflow.start_run() as run:</strong>

        # Log Model using Custom Flavor
        dtw_model = {‘stretch_factor’ : float(ts_stretch_factor),
‘pattern’ : optimal_pattern}
        <strong>mlflow_custom_flavor.log_model(dtw_model, artifact_
path=”model”)</strong>

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

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

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

    return run.info</strong>

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.

www.youtube.com/watch?v=62PAPZo-2ZU

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 pre-loaded 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.

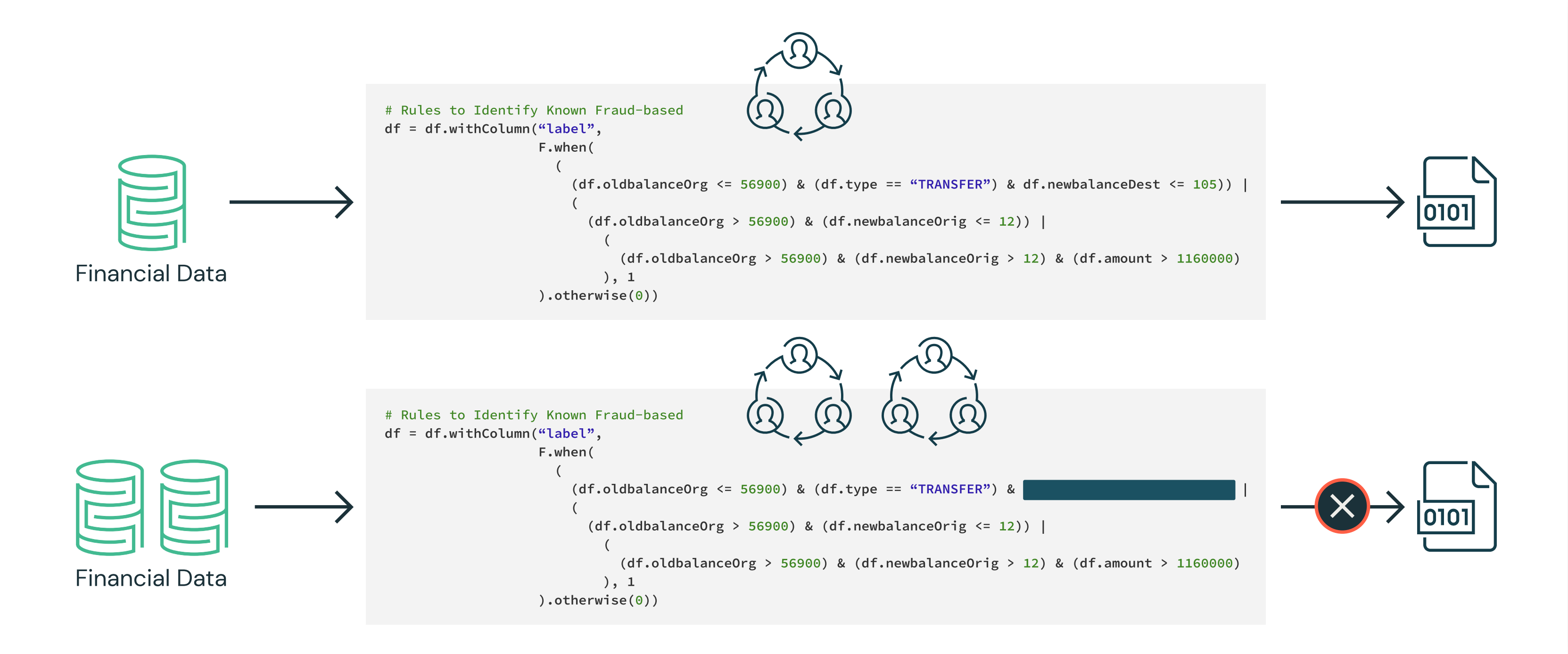
CHAPTER 5:

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

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

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.

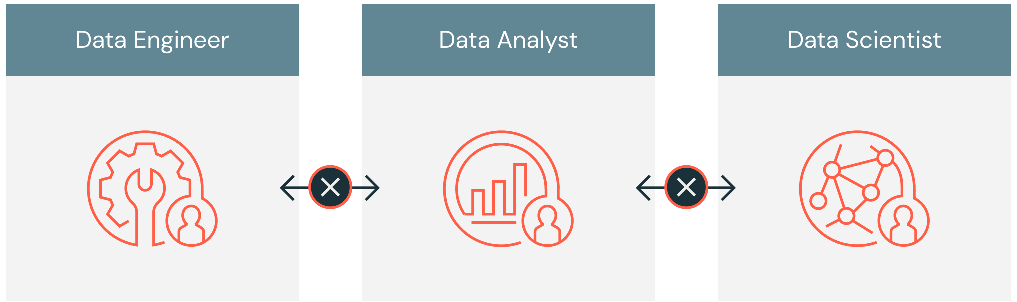


```
# Rules to Identify Known Fraud-based
df = df.withColumn("label",
                   F.when(
                     (
                       (df.oldbalanceOrg <= 56900) & (df.type == "TRANSFER") & df.newbalanceDest <= 105)) |
                       (
                         (df.oldbalanceOrg > 56900)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

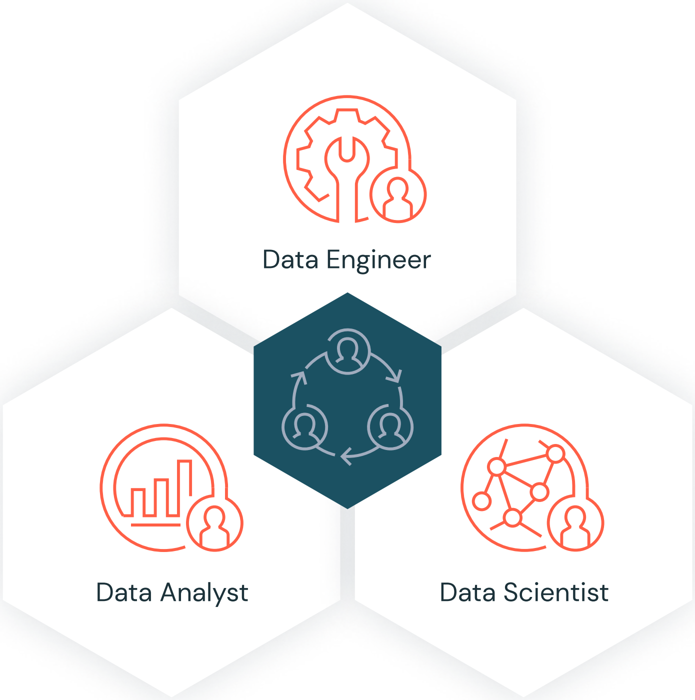
```
# Rules to Identify Known Fraud-based
df = df.withColumn("label",
                   F.when(
                     (
                       (df.oldbalanceOrg <= 56900) & (df.type == "TRANSFER") & [redacted] |
                       (
                         (df.oldbalanceOrg > 56900) & (df.newbalanceOrig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

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.

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.



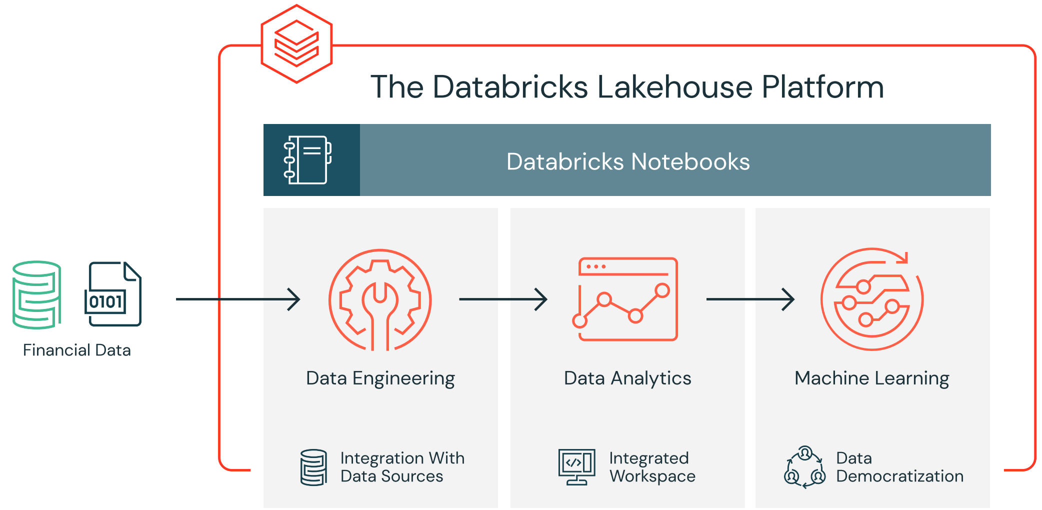
Additionally, the systems in the workflow described above are often siloed, with the domain experts, data scientists and data engineers all compartmentalized. The data engineer is responsible for maintaining massive amounts of data and translating the work of the domain experts and data scientists into 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.

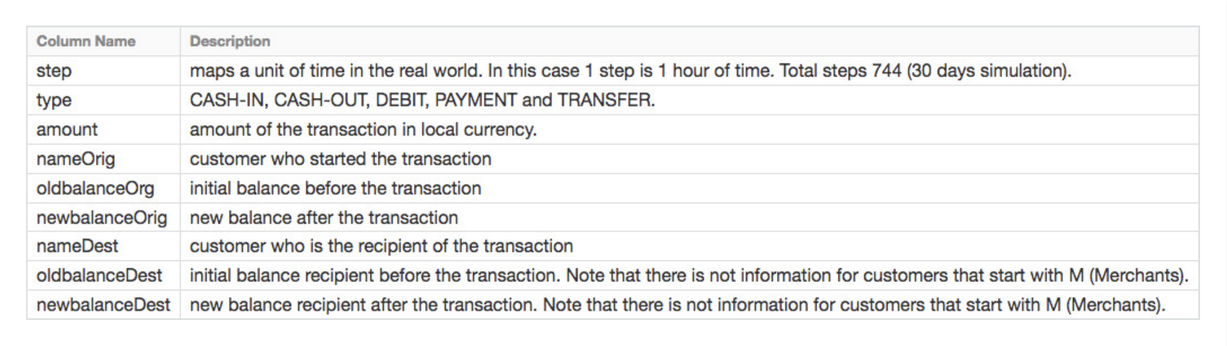
Furthermore, the concern with machine learning models being difficult to interpret may be further assuaged if a decision tree model is used as the initial machine learning model. Because the model is being trained to a set of rules, the decision tree is likely to outperform any other machine learning 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).

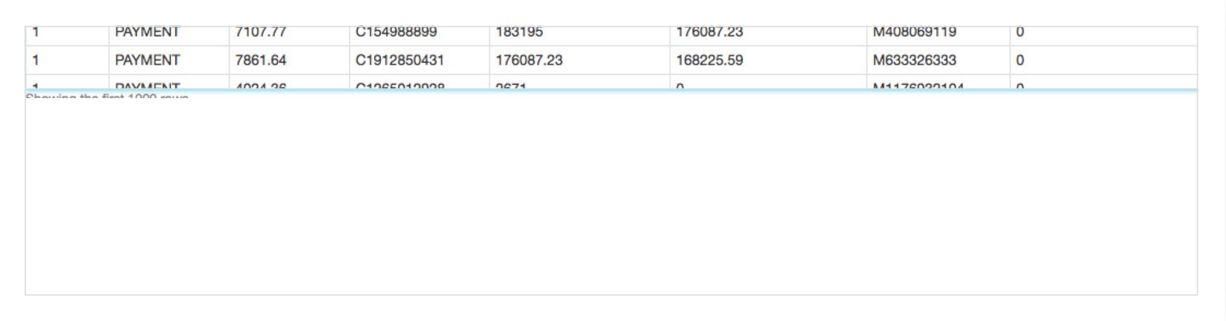
Exploring the data

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)
```



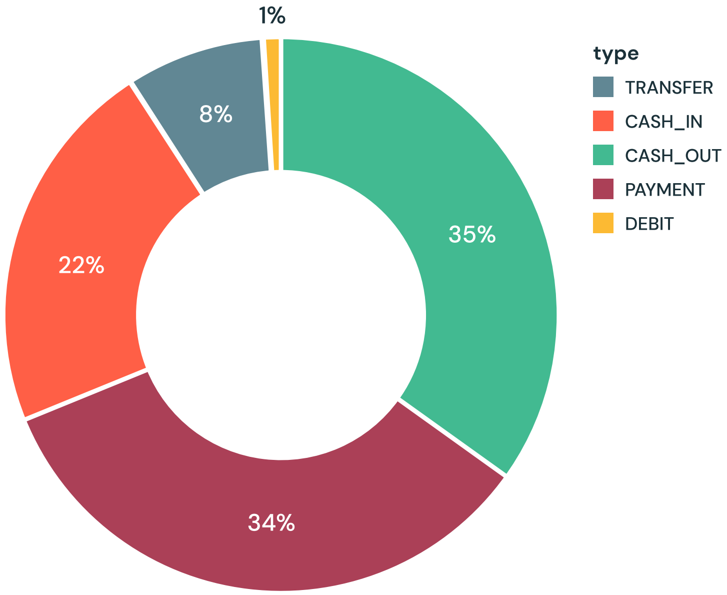
1	PAYMENT	7107.77	C154988899	183195	176087.23	M408069119	0
1	PAYMENT	7861.64	C1912850431	176087.23	168225.59	M633326333	0
1	PAYMENT	4024.36	C1265012928	2671	0	M1176932104	0

Showing the first 1000 rows.

Types of transactions

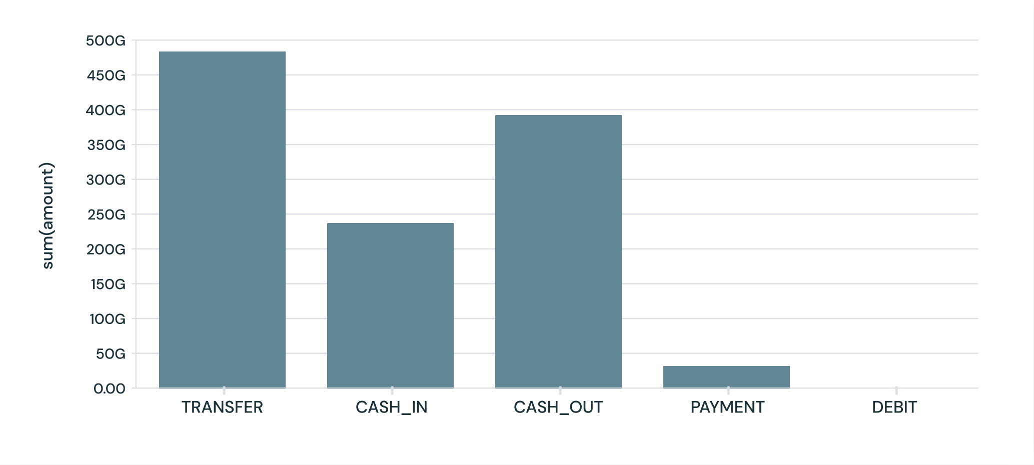
Let's visualize the data to understand the types of transactions the data captures and their contribution to the overall transaction volume.

```
%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
```



Rule-based model

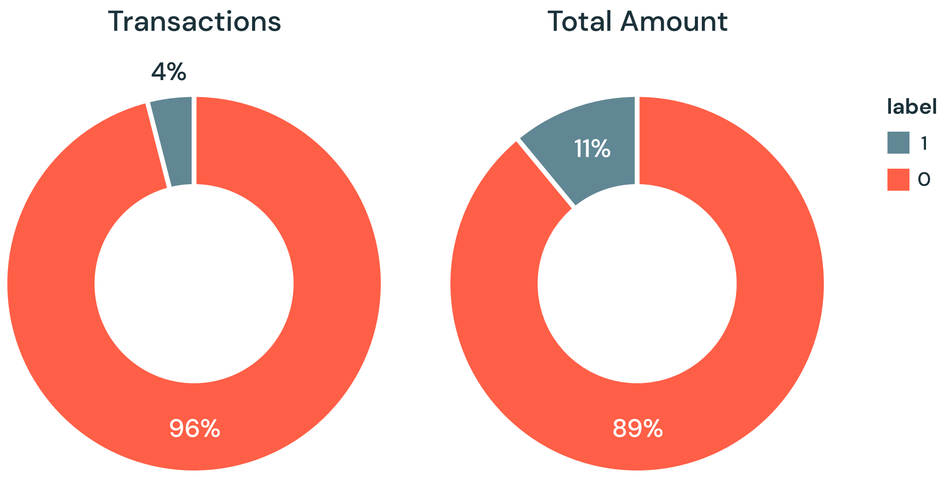
We are not likely to start with a large data set of known fraud cases to train our model. In most practical applications, fraudulent detection patterns are identified by a set of rules established by the domain experts. Here, we create a column called `label` based on these rules.

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

Visualizing data flagged by rules

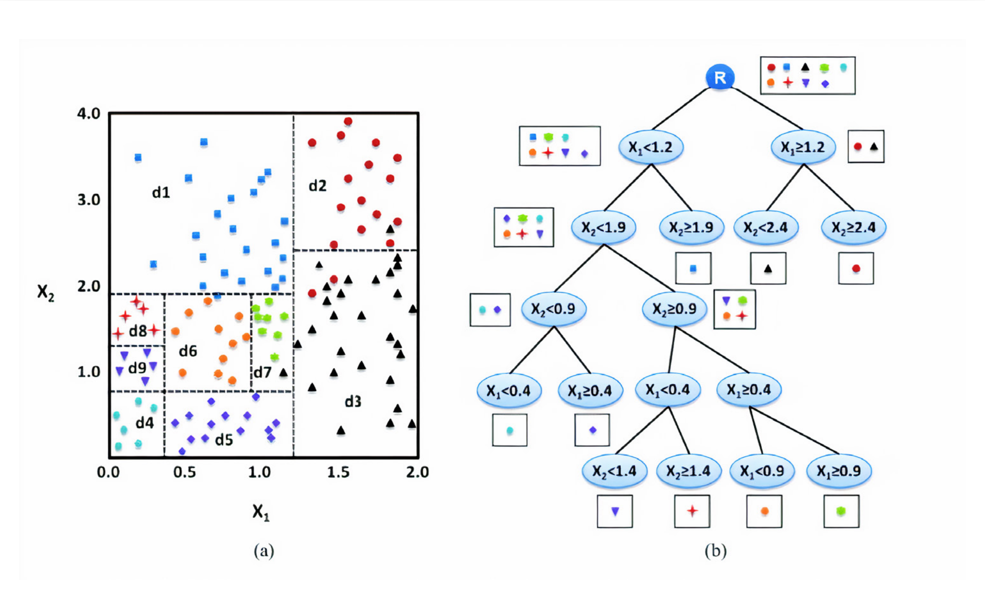
These rules often flag quite a large number of fraudulent cases. Let's visualize the number of flagged transactions. We can see that the rules flag about 4% of the cases and 11% of the total dollar amount as fraudulent.

```
%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.



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

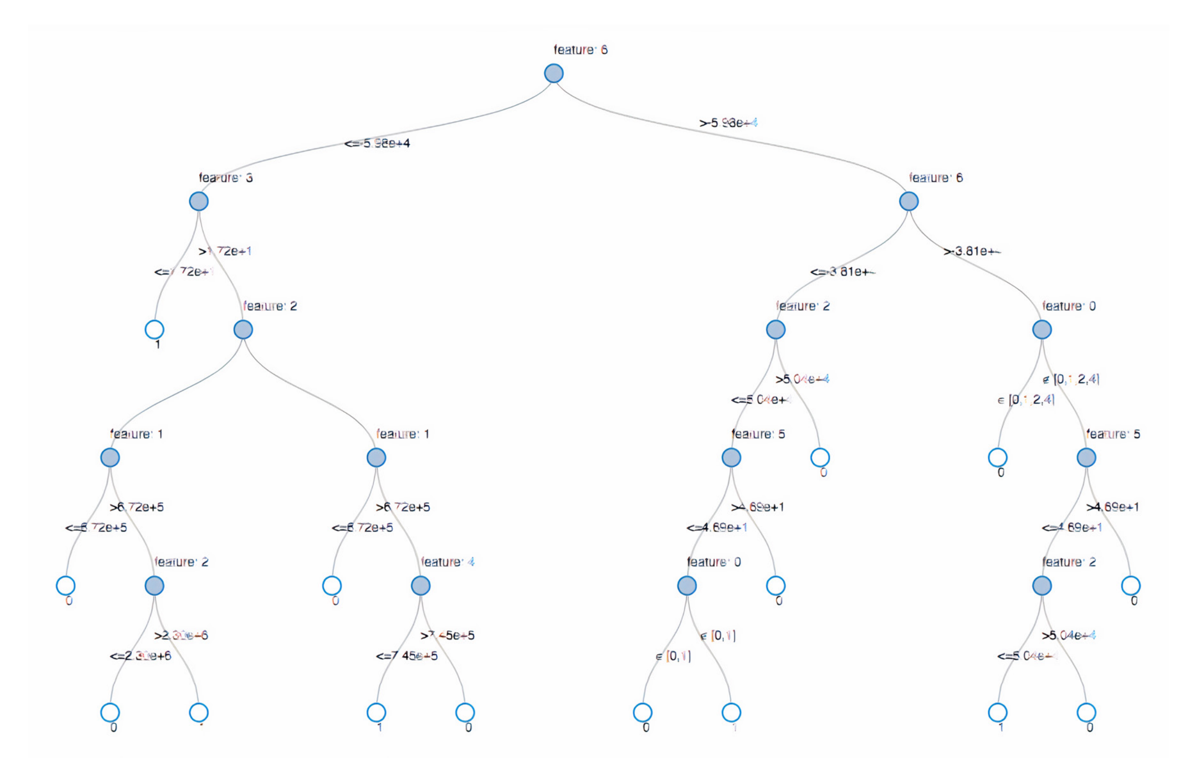
To prepare the data for the model, we must first convert categorical variables to numeric using `.StringIndexer`. We then must assemble all of the features we would like for the model to use. We create a pipeline to contain these feature preparation steps in addition to the decision tree model so that 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.

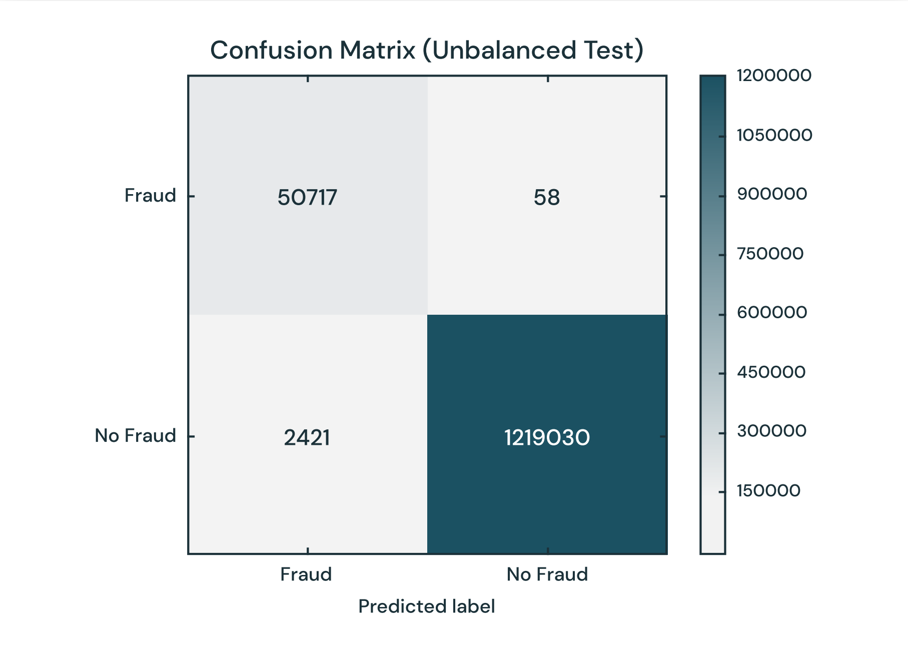
```
<b>from</b> pyspark.ml.tuning <b>import</b> CrossValidator,
ParamGridBuilder

# 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
```

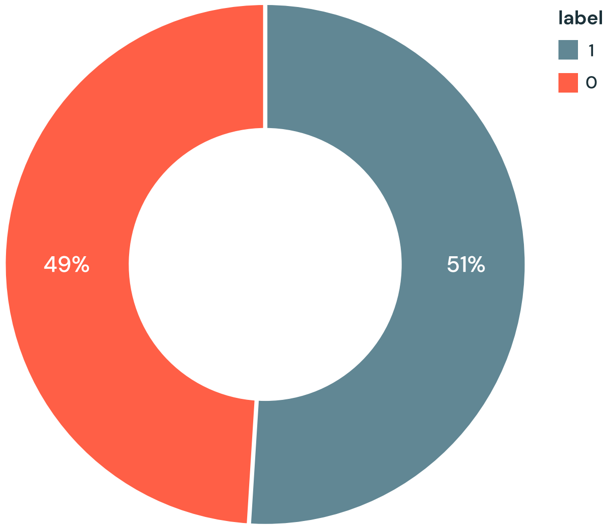
To see how the model misclassified the results, let’s use Matplotlib and pandas to visualize our confusion matrix.



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(<b>False</b>, 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: 0.04024541125893201б
# Balanced training dataset count: 401898
---

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



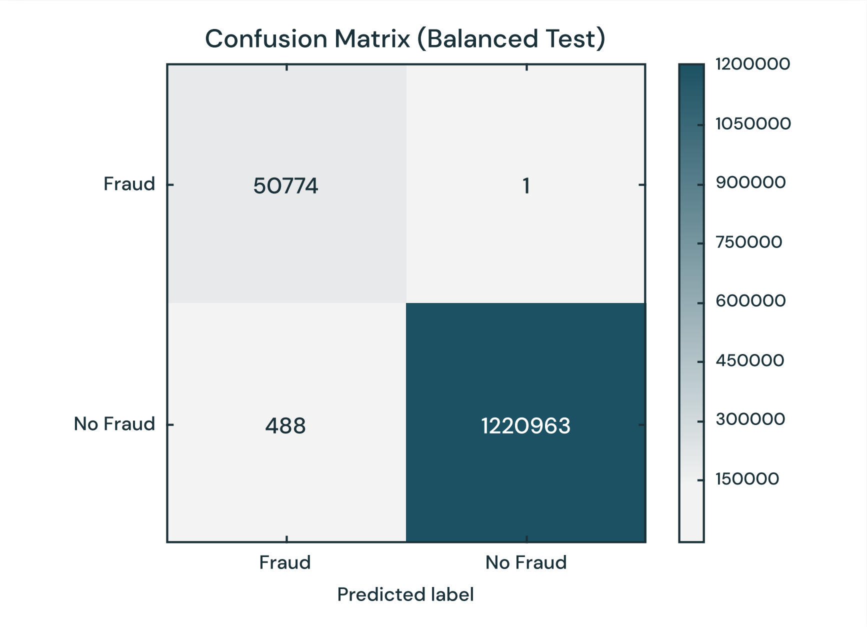
Updating the pipeline

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.

> www.youtube.com/watch?v=x_4S9r-Kks8
>
> www.youtube.com/watch?v=BVISypymHzw

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 helps 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 6:

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

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

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 chapter, we'll discuss the importance of time series forecasting, visualize some sample time series data, and then build a simple model to show the use of Facebook Prophet. Once you're comfortable building a single model, we'll combine Facebook Prophet with the magic of Spark to show you how to train hundreds of models at once, allowing you 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

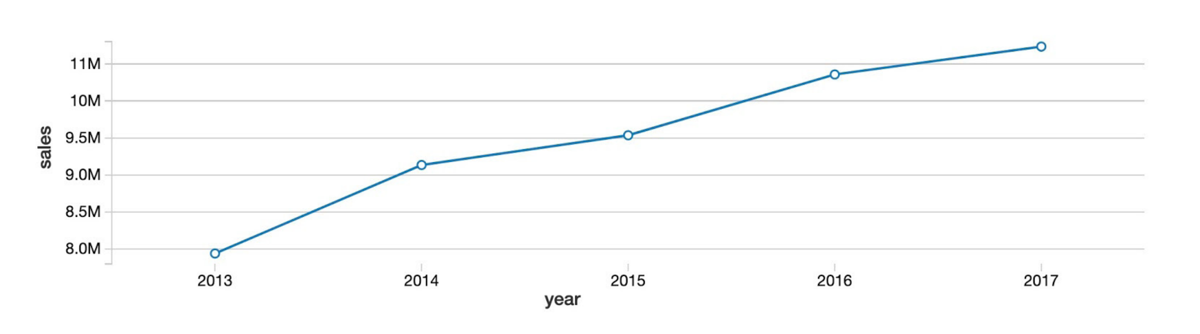
New capabilities, such as those provided by Facebook Prophet, are emerging from the data science community, and companies are seeking the flexibility to apply these machine learning models to their time series forecasting needs.

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 ML models in a timely manner. Luckily, we can use Spark to distribute the training of these models, making it possible to predict both demand for products and services and the unique demand for each product in each location.

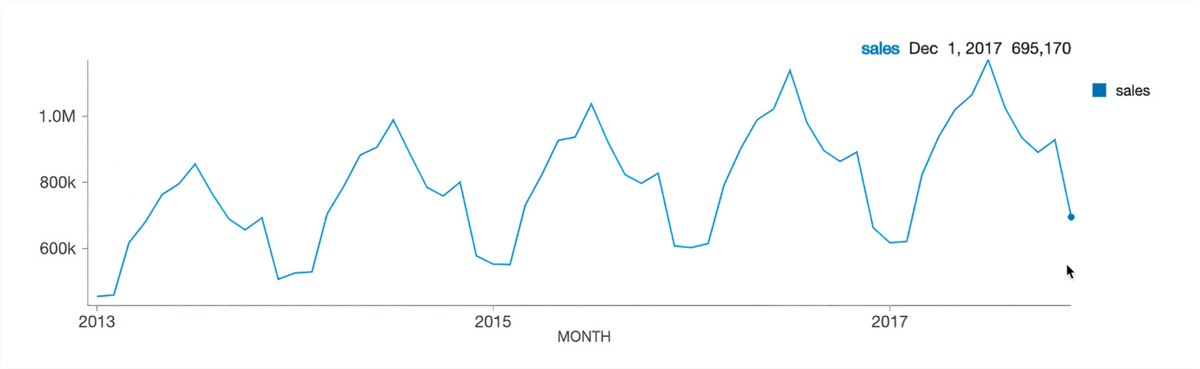
Visualizing demand seasonality in time series data

To demonstrate the use of Prophet to generate fine-grained 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.

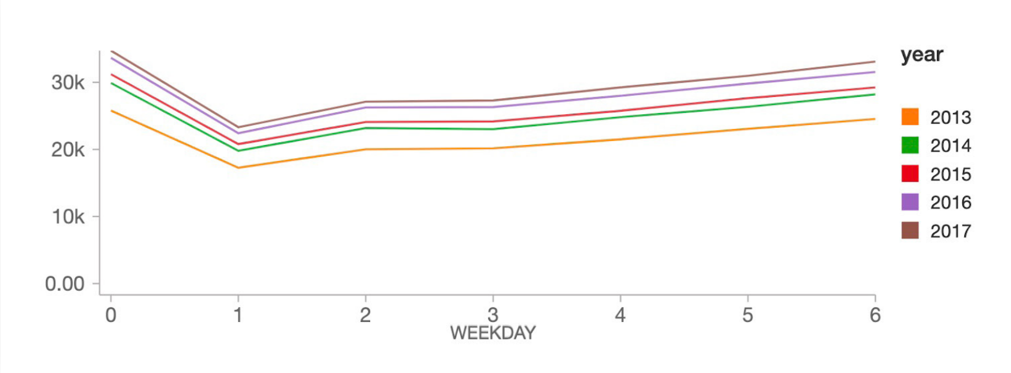
To get started, let's look at the overall yearly sales trend for all products and stores. As you can see, total product sales are increasing year over year with no clear sign of convergence around a plateau.



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 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 Facebook 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:

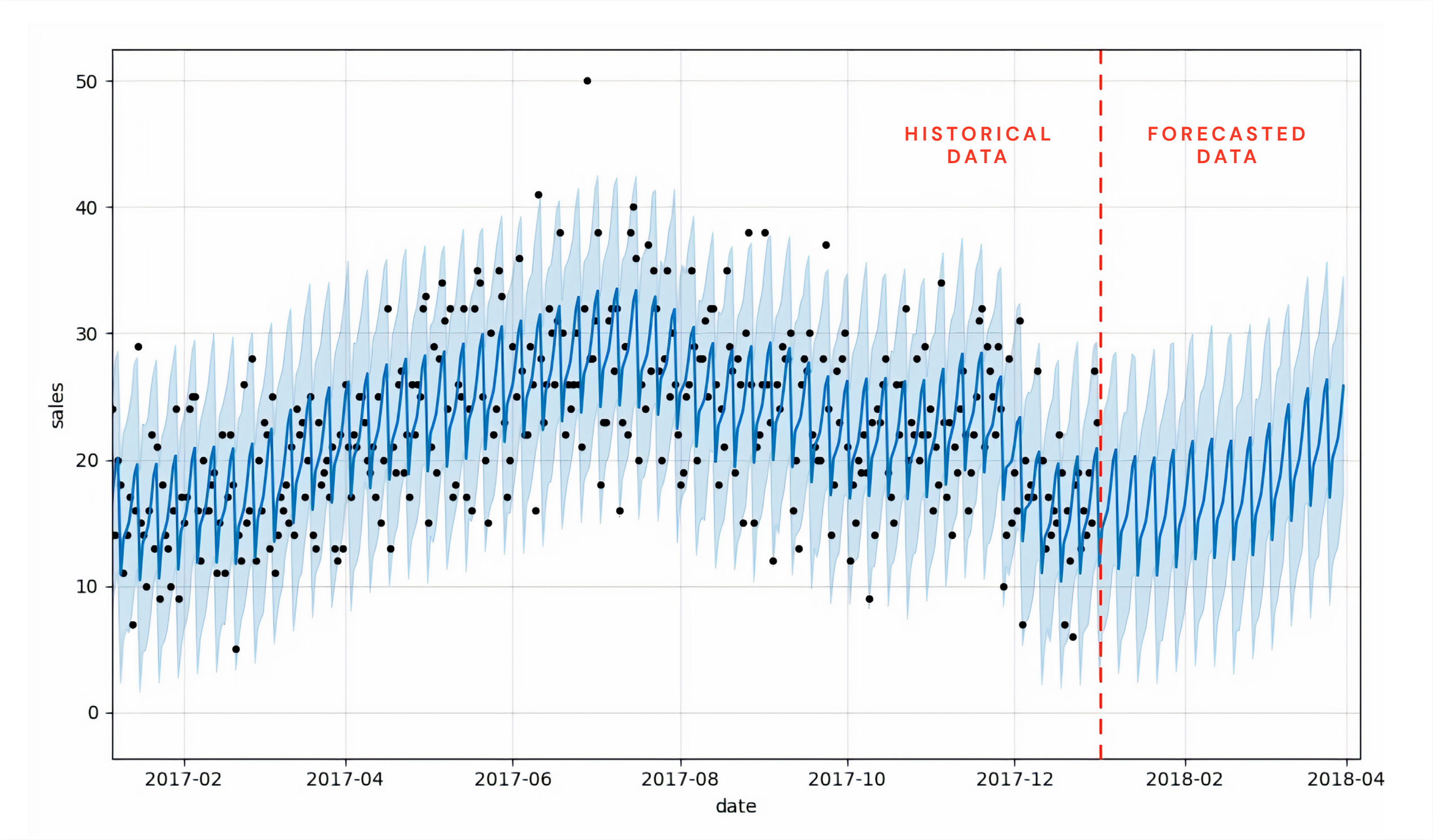
```
import pandas as pd
from fbprophet import Prophet

# 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 model, we can use the power of 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 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 custom pandas function 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 function, which allows us to apply a custom function to each group of data in our DataFrame.

This function 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 abbreviated code below, building our function is relatively straightforward. Unlike in previous versions of Spark, we can declare our functions in a fairly streamlined manner, specifying the type of pandas object we expect to receive and return, i.e., Python type hints.

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.

```
def forecast_store_item(history_pd: pd.DataFrame) -> pd.DataFrame:

    # 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
```

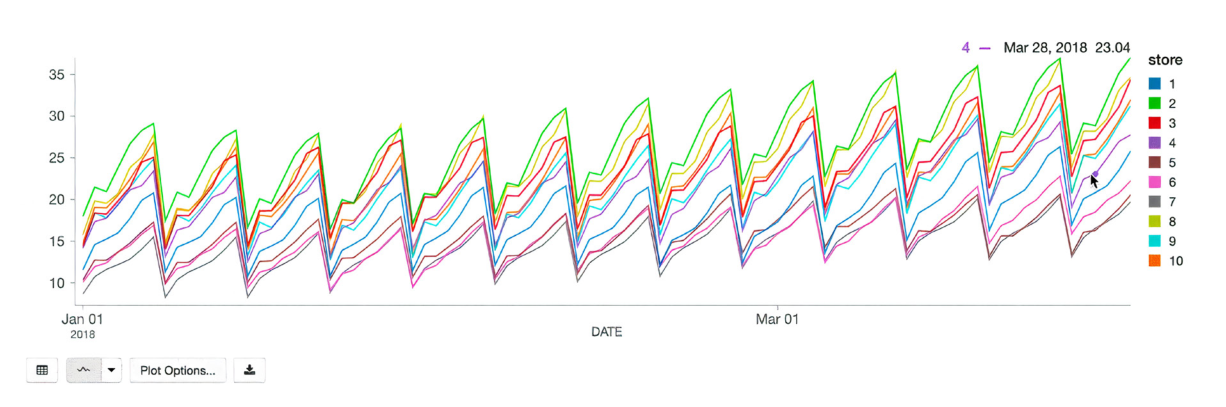
Now, to bring it all together, we use the groupBy command we discussed earlier to ensure our data set is properly partitioned into groups representing specific store and item combinations. We then simply add the applyInPandas function to our DataFrame, allowing it to fit a model and make predictions 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 forecast 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 generate these forecasts in your Databricks environment, please import the following notebook: Fine-Grained Demand Forecasting With Spark 3.

To access the prior version of this notebook, built for Spark 2.0, please click this link.

CHAPTER 7:

Applying Image Classification With PyTorch Lightning on Databricks

Introduction

PyTorch Lightning is a great way to simplify your PyTorch code and bootstrap your deep learning workloads. Scaling your workloads to achieve timely results with all the data in your lakehouse brings its own challenges, however. This article will explain how this can be achieved and how to efficiently scale your code with Horovod.

Increasingly, companies are turning to deep learning in order to accelerate their advanced machine learning applications. For example, computer vision techniques are used nowadays to improve defect inspection for manufacturing; natural language processing is utilized to augment business processes with chatbots and neural network based recommender systems are used to improve customer outcomes.

Training deep learning models, even with well-optimized code, is a slow process, which limits the ability of data science teams to quickly iterate through experiments and deliver results. As such, it is important to know how to best harness compute capacity in order to scale this up.

In this article we will illustrate how to first structure your codebase for maximum code reuse, then show how to scale this from a small single node instance across to a full GPU cluster. We will also integrate it all with MLflow to provide full experiment tracking and model logging.

Part 1 – Data Loading and Adopting PyTorch Lightning

First, let's start with a target architecture.

Cluster setup

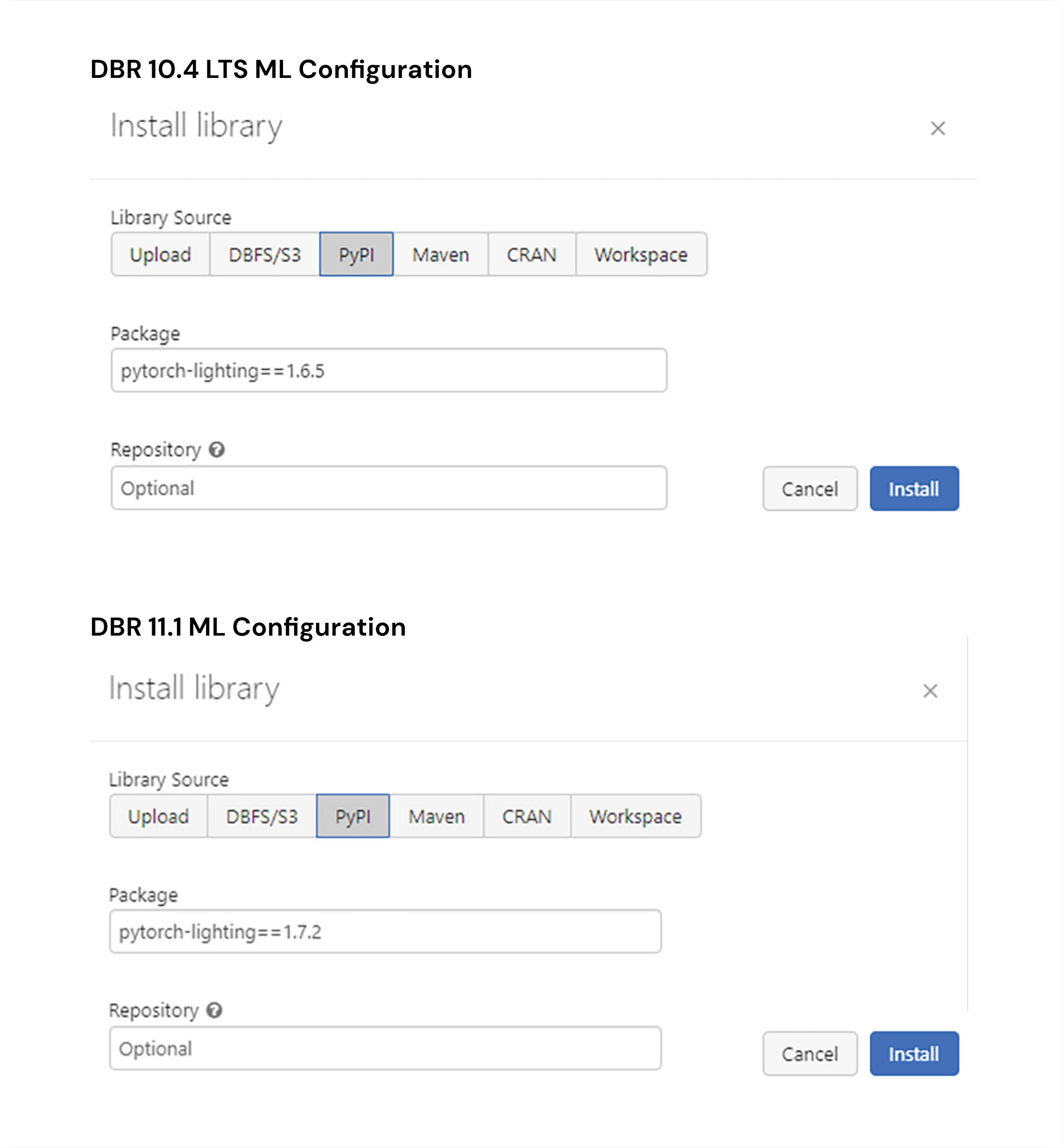
When scaling deep learning, it is important to start small and gradually scale up the experiment in order to efficiently utilize expensive GPU resources. Scale up your code to run on multiple GPUs within a single node before looking to scale across multiple nodes to reduce code complexity.

Databricks supports single-node clusters to support this very usage pattern. See: Azure Single Node Clusters, AWS Single Node Clusters, GCP Single Node Clusters. In terms of instance selection, NVIDIA T4 GPUs provide a cost-effective instance type to start with. On AWS these are available in G4 instances. On Azure these are available in NCasT4_v3 instances. On GCP these are available as A2 instances.

To follow through the notebooks, an instance type with at least 64GB RAM is required. The modeling process is memory intensive and it is possible to run out of RAM with smaller instances, which can result in the following error.

```
Fatal error: The Python kernel is unresponsive.
```

The code was built and tested on Databricks Runtime 10.4 LTS for Machine Learning and also 11.1 ML. On DBR 10.4 LTS ML only pytorch-lightning up to 1.6.5 is supported. On DBR 11.1 ML, pytorch-lightning 1.7.2 has been tested. We have installed our libraries as workspace level libraries. Unlike using %pip, which installs libraries only for the active notebook on the driver node, workspace libraries are installed on all nodes, which we will need later for distributed training.



DBR 10.4 LTS ML Configuration

Install library

Library Source: Upload | DBFS/S3 | **PyPI** | Maven | CRAN | Workspace

Package: pytorch-lighting==1.6.5

Repository: Optional

DBR 11.1 ML Configuration

Install library

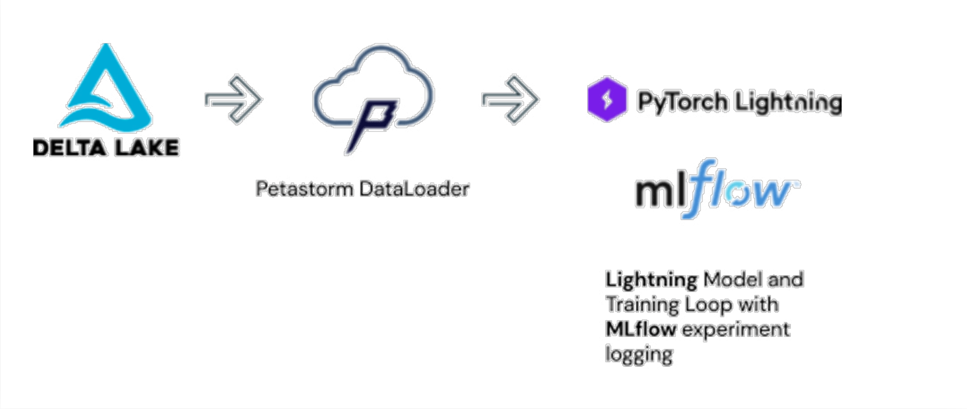
Library Source: Upload | DBFS/S3 | **PyPI** | Maven | CRAN | Workspace

Package: pytorch-lighting==1.7.2

Repository: Optional

Figure 1: Library configuration

Target Architecture



DELTA LAKE
Petastorm DataLoader
PyTorch Lightning
mlflow
Lightning Model and Training Loop with MLflow experiment logging

Figure 2: Key components

The goal of this article is to build up a codebase structured as above. We will store our data using the open source Linux Foundation project Delta Lake. Under the hood, Delta Lake stores the raw data in Parquet format. Petastorm takes on the data loading duties and provides the interface between the lakehouse and our deep learning model. MLflow will provide experiment tracking tools and allow for saving out the model to our model registry.

With this setup, we can avoid unnecessary data duplication costs as well as govern and manage the models that we are training.

Part 2 – Example Use Case and Library Overview

Example use case

For this use case example, we will use the TensorFlow flowers data set. This data set will be used for a classification type problem where we are trying to identify which class of flower is which.

Leveraging your data lake for deep learning with Petastorm

Historically, data management systems like lakehouses and data warehouses have developed in parallel with, rather than in integration with, machine learning frameworks. As such, PyTorch DataLoader modules do not support the Parquet format out of the box. They also do not integrate with lakehouse metadata structures like the hive metastore.

The Petastorm project provides the interface between your lakehouse tables and PyTorch. It also handles data sharding across training nodes and provides a caching layer. Petastorm comes prepackaged in the Databricks Runtime for Machine Learning.

Let's first become familiar with the data set and how to work with it. Of note is that all we need to do to transform a Spark DataFrame into a Petastorm object is the code:

```
peta_conv_df = make_spark_converter(preprocessed_df)
```

Once we have the `spark_converter` object, we can convert that into a PyTorch DataLoader using:

```
with peta_conv_df.make_torch_dataloader(transform_spec=transform_func)
as converted_dataset
```

This then provides a converted_dataset DataLoader that we can use in our PyTorch code as per normal.

Open and follow the notebook titled Exploring the flowers dataset. A standard ML runtime cluster will be sufficient; there is no need to run this on a GPU cluster.

Simplify and structure your model — enter PyTorch Lightning

By default, PyTorch code can get quite verbose. There is the model definition, the training loop and the setup of the dataloaders. By default all this code is mixed together, making it hard to swap data sets and models in and out, which can be key for fast experimentation.

PyTorch Lightning helps to make this simpler by greatly reducing the boilerplate required to set up the experimental model and the main training loop. It is an opinionated approach to structuring PyTorch code, which allows for more readable maintainable code.

For our project, we will break up the code into three main modules

- **PyTorch Model**
- **DataLoaders and Transformations**
- **Main Training Loop**

This will help to make our code more portable and also improve organization. These classes and functions will all be pulled into the main execution notebook, via `%run`, where the training hyperparameters will be defined and the code actually executed.

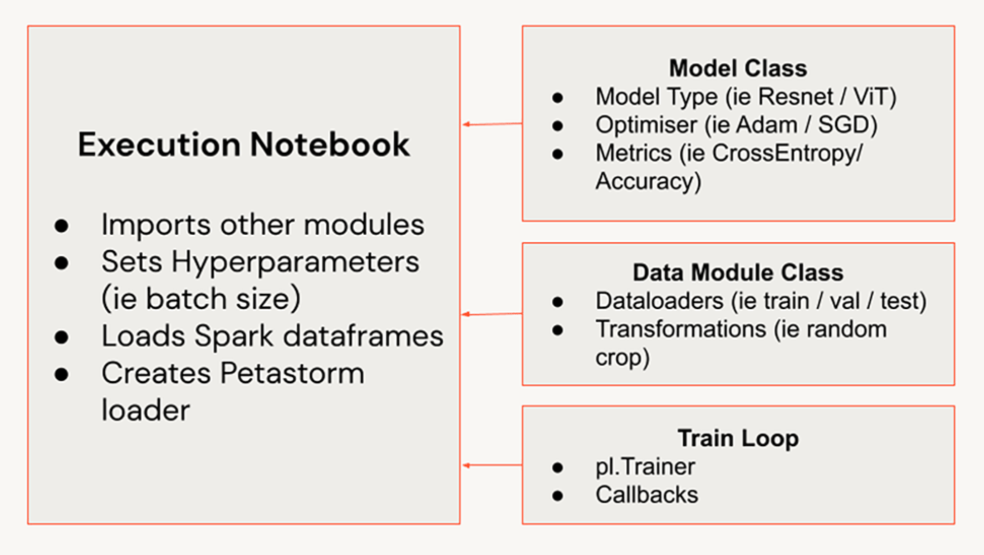

Execution Notebook
Imports other modules
Sets Hyperparameters (ie batch size)
Loads Spark dataframes
Creates Petastorm loader
Model Class
Model Type (ie Resnet / ViT)
Optimiser (ie Adam / SGD)
Metrics (ie CrossEntropy/ Accuracy)
Data Module Class
Dataloaders (ie train / val / test)
Transformations (ie random crop)
Train Loop
pl.Trainer
Callbacks

Figure 3: Code layout

Model definition:

This module contains the code for the model architecture itself in a model class, LightningModule. This is where the model architecture lives. For reference, this is the module that needs updating to leverage popular model frameworks like timm, HuggingFace and the like. This module will also contain the definitions for optimizers. In this case, we just use SGD, but it can be parameterized to test out other types of optimizers.

DataLoader class:

Unlike with native PyTorch, where DataLoader code is intermixed with the model code, PyTorch Lightning allows us to split it out into a separate LightningDataModule class. This allows for easier management of data sets and the ability to quickly test different interactions of your data sets.

When building a `LightningDataModule` with a Petastorm DataLoader, we feed in the spark_converter object rather than the raw `spark dataframes`. The Spark DataFrame is managed by the underlying Spark cluster, which is already distributed, whereas the PyTorch DataLoader will be distributed through other means later.

Main training loop:

This is the main training function. It takes the `LightningDataModule` and the `LightningModule` defining the model before feeding it into the `Trainer` class. We will instantiate the PyTorch Lightning Trainer and define all necessary callbacks here.

As we scale up the training process later on, we do not need some processes like MLflow logging to be run on all the processing nodes. As such, we will restrict these to run on the first GPU only.

```
if device_id == 0:

    # we only need this on node 0
    mlflow.pytorch.autolog()
```

Checkpointing our model during training is important for preserving progress, but PyTorch Lighting will by default handle this for us and we do not need to add code.

Follow along in the Building the PyTorch Lightning Modules notebook.

Part 3 – Scaling the Training Job

While single-GPU training is much faster than CPU training, it is often not enough. Proper production models can be large and the data sets required to train these properly will be large too. Hence, we need to look into how we can scale our training across multiple GPUs.

The main approach to distributing deep learning models is via data parallelism, where we send a copy of the model to each GPU and feed in different shards of data to each. This lets us increase the batch size and leverage higher learning rates to improve training times as discussed in this article.

To assist us in distributing the training job across GPUs, we can leverage Horovod. Horovod is another Linux Foundation project that offers us an alternative to manually triggering distributed PyTorch processes across multiple nodes. Databricks Runtime for Machine Learning includes by default the HorovodRunner class, which helps us scale on both single-node and multi-node training.

In order to leverage Horovod, we need to create a new “super” train loop.

```
def train_hvd():
  hvd.init()

  # MLflow setup for the worker processes
  mlflow.set_tracking_uri("databricks")
  os.environ['DATABRICKS_HOST'] = db_host
  os.environ['DATABRICKS_TOKEN'] = db_token


  hvd_model = LitClassificationModel(class_count=5, learning_rate=1e-
5*hvd.size(), device_id=hvd.rank(), device_count=hvd.size())
  hvd_datamodule = FlowersDataModule(train_converter, val_converter,
device_id=hvd.rank(), device_count=hvd.size())

  # `gpus` parameter here should be 1 because the parallelism is
controlled by Horovod
  return train(hvd_model, hvd_datamodule, gpus=1, strategy="horovod",
device_id=hvd.rank(), device_count=hvd.size())
```

This function will start Horovod `hvd.init()` and ensure that our DataModule and train function are triggered with the correct node number `hvd.rank()` and total number of devices `hvd.size()`. As discussed in this Horovod article we scale up the learning rate with the number of GPUs.

```
hvd_model = LitClassificationModel(class_count=5, learning_rate=1e-
5*hvd.size(), device_id=hvd.rank(), device_count=hvd.size())
```

Then we return the normal train loop with the GPU count set to 1 as Horovod is handling the parallelism.

Follow along in the Main Execution notebook and we will go through the ways to go from single- to multi-GPU.

Step 1 – Scaling on one node

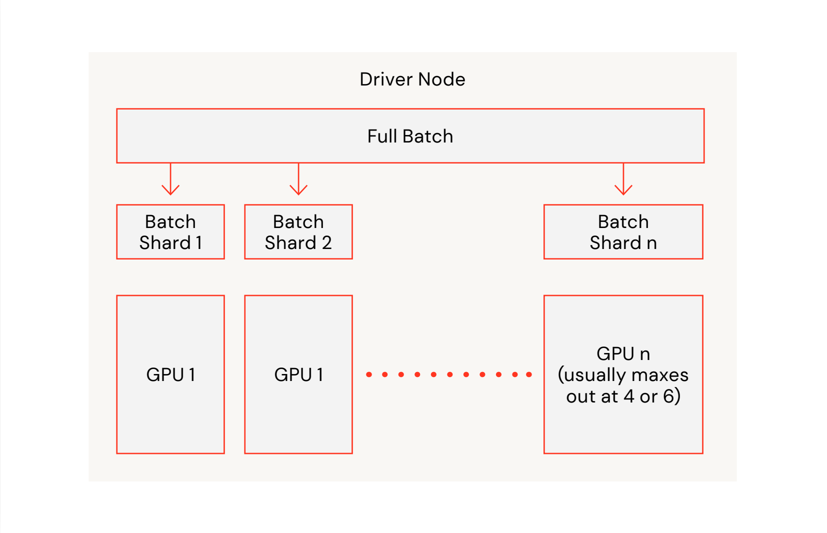


Figure 4: Single-node scaling

Scaling on one node is the easiest way to scale. It is also very performant, as it avoids the network traffic required for multi-node training. Unlike Spark-native ML Libraries, most deep learning training processes do not automatically recover from node failures. PyTorch Lightning, however, does automatically save out checkpoints for recovering training epochs.

In our code, we set the `default_dir` parameter to a DBFS location in the train function. This is where PyTorch Lightning will save out the checkpoints. If we set a `ckpt_restore` path to point to ckpt, the train function will resume training from that checkpoint.

```
def train(model, dataloader, gpus:int=0,
          strategy:str=None, device_id:int=0,
          device_count:int=1, logging_level=logging.INFO,
          default_dir:str=’/dbfs/tmp/trainer_logs’,
          ckpt_restore:str=None,

          mlflow_experiment_id:str=None):
```

To scale out our train function to multiple GPUs on one node, we will use `HorovodRunner`:

```
from sparkdl import HorovodRunner

hr = HorovodRunner(np=-4, driver_log_verbosity=’all’)
hvd_model = hr.run(train_hvd)
```

Setting `np` to negative will make it run on the single driver node with 4 GPUs. A positive `np` value will spread the training across other worker nodes.

Step 2 – Scaling across nodes

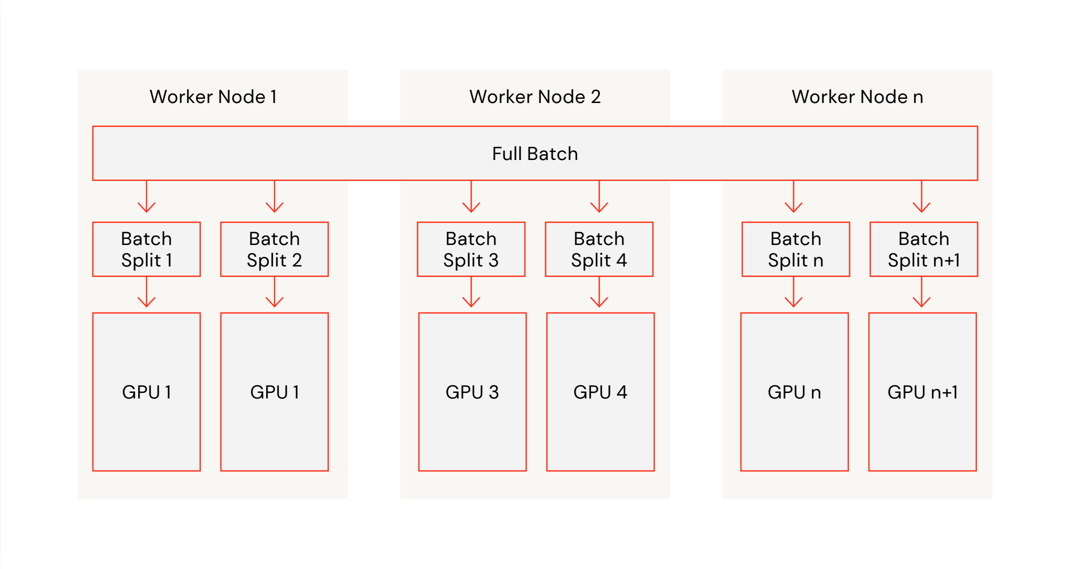
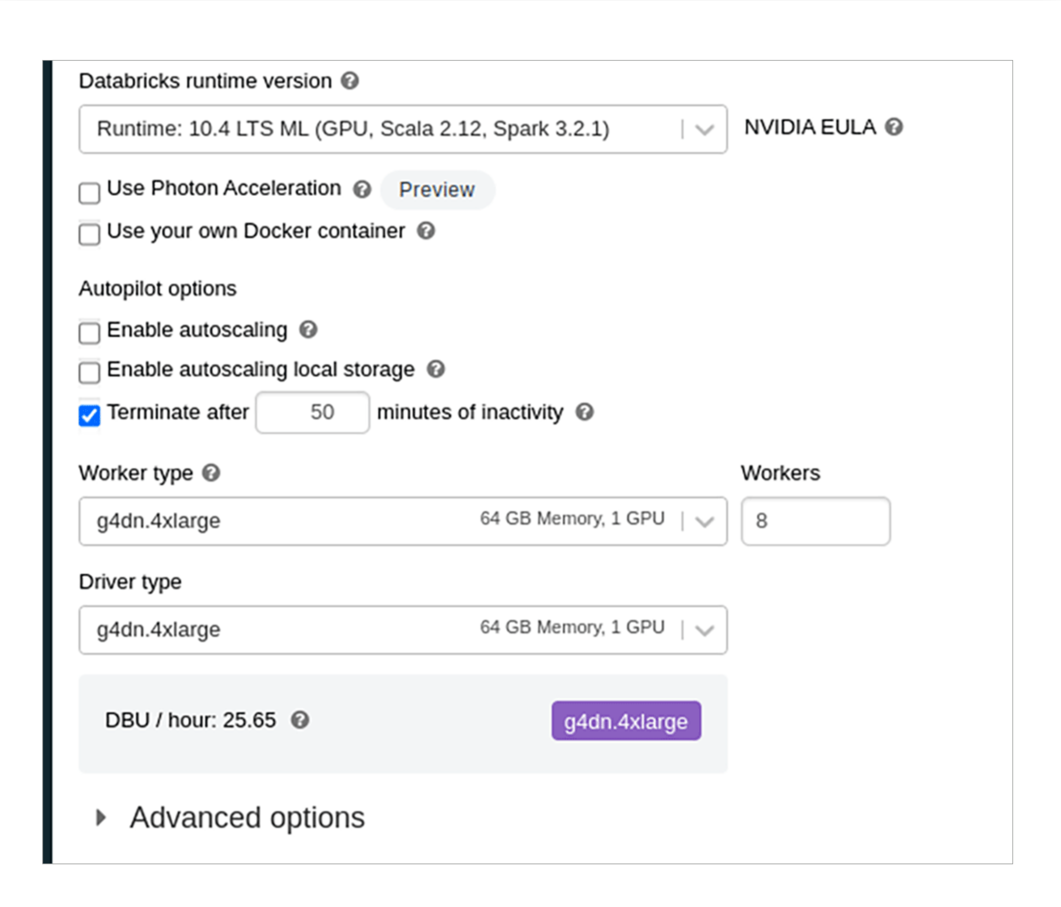


Figure 5: Multi-node scaling

We have already wrapped our training function with a Horovod wrapper and we have already successfully leveraged HorovodRunner for single-node multi-GPU processing. The final step is to go to a multi-node/multi-GPU setup. If you have been following along with a single-node cluster, this is the point where we will move to a multi-node cluster. For the code that follows, we will use the cluster configuration shown at right:



Databricks runtime version
Runtime: 10.4 LTS ML (GPU, Scala 2.12, Spark 3.2.1)
NVIDIA EULA

□ Use Photon Acceleration Preview
□ Use your own Docker container

Autopilot options

□ Enable autoscaling
□ Enable autoscaling local storage
☑ Terminate after 50 minutes of inactivity

Worker type: g4dn.4xlarge 64 GB Memory, 1 GPU
Workers: 8

Driver type: g4dn.4xlarge 64 GB Memory, 1 GPU

DBU / hour: 25.65 g4dn.4xlarge

Advanced options

Figure 6: Multi-node cluster setup

When running distributed training on Databricks, autoscaling is not currently supported, so we will set our workers to a fixed number ahead of time.

```
hr = HorovodRunner(np=8, driver_log_verbosity=’all’)
hvd_model = hr.run(train_hvd)
```

A common problem that will occur as you scale up your distributed deep learning job is that the Petastorm table has not been partitioned well enough to ensure that all the GPUs get a batch split. We need to make sure that we have at least as many data partitions as we have GPUs.

We address this in our code by setting the number of GPUs in the `prepare_data` function with the `num_devices` variable.

```
flowers_df, train_converter, val_converter = prepare_data(data_dir=Data_
Directory, num_devices=NUM_DEVICES)

datamodule = FlowersDataModule(train_converter=train_converter,
                               val_converter=val_converter)
```

This simply calls a standard Spark repartition command. We set the number of partitions to be a multiple of the `num_devices`, the number of GPUs, to make sure that the data set has sufficient partitions for all the GPUs we have allocated for the training process. Insufficient partitions is a common cause of idling GPUs.

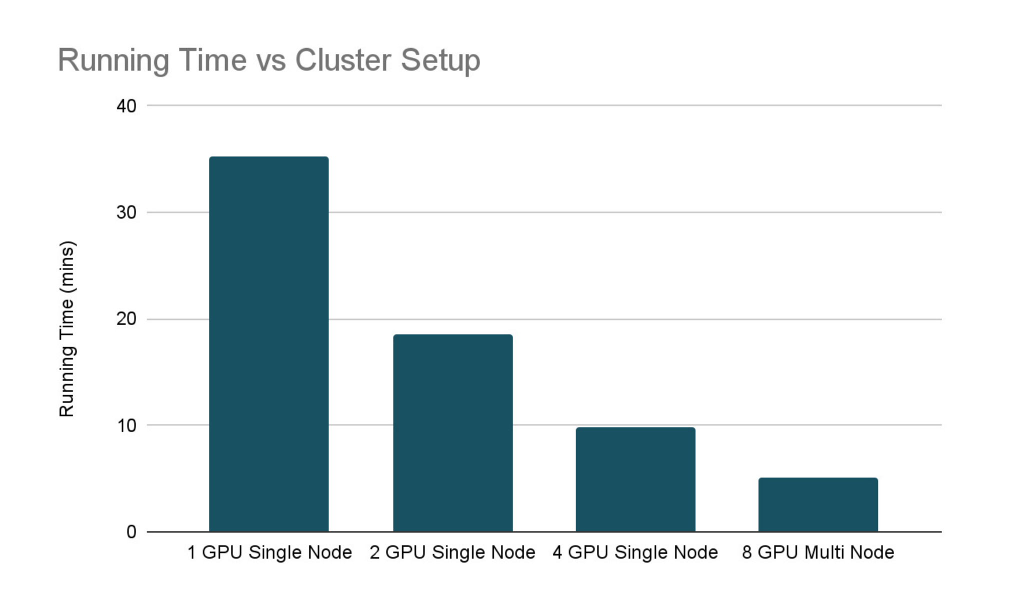
```
flowers_dataset = flowers_dataset.repartition(num_devices*2)
```

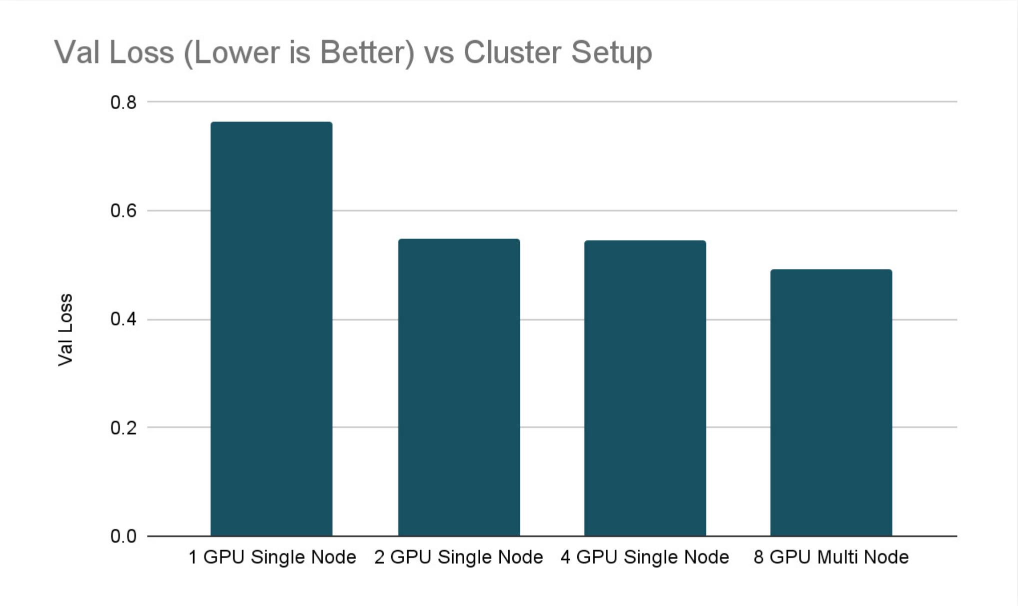
Analysis

When training deep neural networks, it is important to make sure we do not overfit the network. The standard way to manage this is to leverage early stopping. This process checks to make sure that with each epoch, we are still seeing improvements to the metric that we set it to monitor. In this case, `val_loss`.

For our experiments, we set `min_delta` to 0.01, so we expect to see at least 0.01 improvement to `val_loss` each epoch. We set `patience` to be 10 so the train loop will continue to run up to 10 epochs of no improvement before the training stops. We set this to make sure that we can eke out the last drop of performance. To keep the experimentation shorter, we also set a `stopping_threshold` of 0.55 so we will stop the training process once our `val_loss` drops below this level.

With those parameters in mind, the results of our scaling experiments are as follows:





As we can see, in the Running Time vs Cluster Setup chart, we nearly halved the training time as we increased the system resources. The scaling is not quite linear, which is due to the overhead of coordinating the training process across different GPUs. When scaling deep learning, it is common to see diminishing returns and hence it is important to make sure that the train loop is efficient prior to adding GPUs.

That is not the full picture, however, as per the best practices advised in our previous blog article, How (Not) To Scale Deep Learning in 6 Easy Steps, we used `EarlyStopping` hence it is important to check the final validation loss achieved by the various training runs as well. In this case, we set the `stopping_threshold` of 0.55. Interestingly, the single-GPU setup stopped at a worse validation loss than the multi-GPU setups. The single-GPU training ran till there were no more improvements in the `val_loss`.

Get started

We have shown how you can leverage PyTorch Lightning within Databricks and wrap it with the `HorovodRunner` to scale across multiple nodes, as well as provided some guidance on how to leverage `EarlyStopping`. Now it's your turn to try.

Notebooks:

Exploring the flowers dataset →

Building the PyTorch Lightning Modules →

Main Execution Notebook →

See Also:

HorovodRunner →

Petastorm →

Deep Learning Best Practices →

How (Not) to Scale Deep Learning →

Leveling the Playing Field: HorovodRunner for Distributed Deep Learning Training →

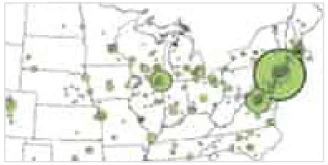
CHAPTER 8:

Processing Geospatial Data at Scale With Databricks

By **Nima Razavi** and **Michael Johns**

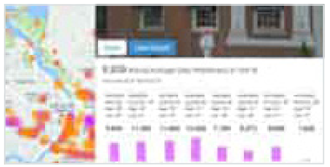
The evolution and convergence of technology has fueled a vibrant marketplace for timely and accurate geospatial data. Every day, billions of handheld and IoT devices along with thousands of airborne and satellite remote sensing platforms generate hundreds of exabytes of location-aware data. This boom of geospatial big data combined with advancements in machine learning is enabling organizations across industries to build new products and capabilities.

FRAUD AND ABUSE



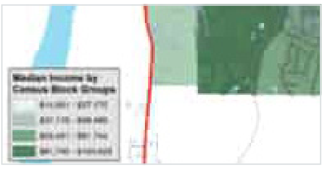
Detect patterns of fraud and collusion (e.g., claims fraud, credit card fraud)

RETAIL



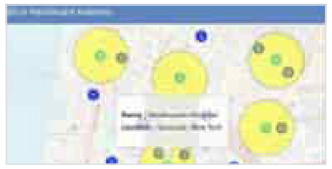
Site selection, urban planning, foot traffic analysis

FINANCIAL SERVICES



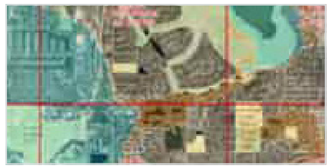
Economic distribution, loan risk analysis, predicting sales at retail, investments

HEALTHCARE



Identifying disease epicenters, environmental impact on health, planning care

DISASTER RECOVERY

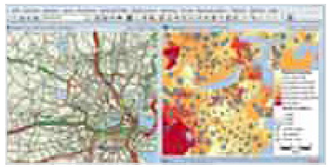


Flood surveys, earthquake mapping, response planning

DEFENSE AND INTEL

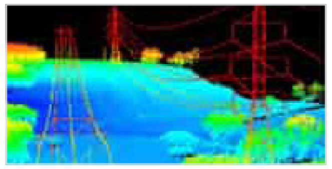
Reconnaissance, threat detection, damage assessment

INFRASTRUCTURE



Transportation planning, agriculture management, housing development

ENERGY



Climate change analysis, energy asset inspection, oil discovery

Maps leveraging geospatial data are used widely across industries, spanning multiple use cases, including disaster recovery, defense and intel, infrastructure and health services.

For example, numerous companies provide localized drone-based services such as mapping and site inspection (reference Developing for the Intelligent Cloud and Intelligent Edge). Another rapidly growing industry for geospatial data is autonomous vehicles. Startups and established companies alike are amassing large corpuses of highly contextualized geodata from vehicle sensors to deliver the next innovation in self-driving cars (reference Databricks fuels wejo's ambition to create a mobility data ecosystem). Retailers and government agencies are also looking to make use of their geospatial data. For example, foot-traffic analysis (reference Building Foot-Traffic Insights Data Set) can help determine the best location to open a new store or, in the public sector, improve urban planning. Despite all these investments in geospatial data, a number of challenges exist.

Challenges analyzing geospatial at scale

The first challenge involves dealing with scale in streaming and batch applications. The sheer proliferation of geospatial data and the SLAs required by applications overwhelms traditional storage and processing systems. Customer data has been spilling out of existing vertically scaled geodatabases into data lakes for many years now due to pressures such as data volume, velocity, storage cost and strict schema-on-write enforcement. While enterprises have invested in geospatial data, few have the proper technology architecture to prepare these large, complex data sets for downstream analytics. Further, given that scaled data is often required for advanced use cases, the majority of AI-driven initiatives are failing to make it from pilot to production.

Compatibility with various spatial formats poses the second challenge. There are many different specialized geospatial formats established over many decades as well as incidental data sources in which location information may be harvested:

- Vector formats such as GeoJSON, KML, shapefile and WKT
- Raster formats such as ESRI Grid, GeoTIFF, JPEG 2000 and NITF
- Navigational standards such as used by AIS and GPS devices
- Geodatabases accessible via JDBC/ODBC connections such as PostgreSQL/PostGIS
- Remote sensor formats from hyperspectral, multispectral, lidar and radar platforms
- OGC web standards such as WCS, WFS, WMS and WMTS
- Geotagged logs, pictures, videos and social media
- Unstructured data with location references

In this blog post, we give an overview of general approaches to deal with the two main challenges listed above using the Databricks Unified Data Analytics Platform. This is the first part of a series of blog posts on working with large volumes of geospatial data.

Scaling geospatial workloads with Databricks

Databricks offers a unified data analytics platform for big data analytics and machine learning used by thousands of customers worldwide. It is powered by Apache Spark™, Delta Lake and MLflow with a wide ecosystem of third-party and available library integrations. Databricks UDAP delivers enterprise-grade security, support, reliability and performance at scale for production workloads. Geospatial workloads are typically complex, and there is no one library fitting all use cases. While Apache Spark does not offer geospatial Data Types natively, the open source community as well as enterprises have directed much effort to develop spatial libraries, resulting in a sea of options from which to choose.

There are generally three patterns for scaling geospatial operations such as spatial joins or nearest neighbors:

1. Using purpose-built libraries that extend Apache Spark for geospatial analytics. GeoSpark, GeoMesa, GeoTrellis and RasterFrames are a few of such libraries used by our customers. These frameworks often offer multiple language bindings and have much better scaling and performance than non-formalized approaches, but can also come with a learning curve.

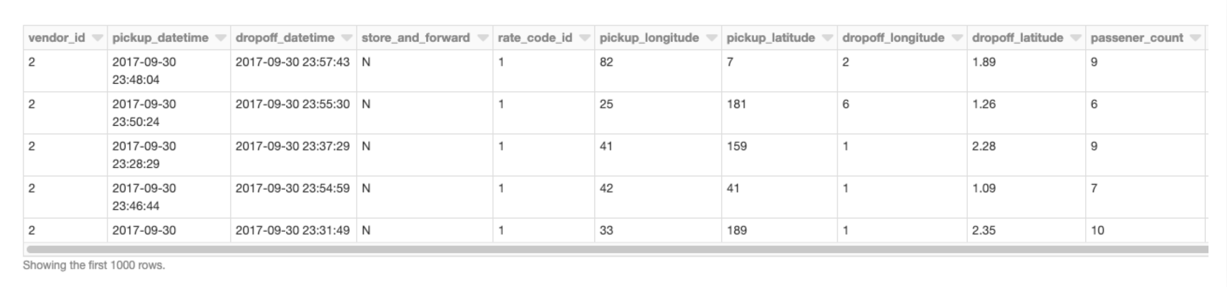
2. Wrapping single-node libraries such as GeoPandas, Geospatial Data Abstraction Library (GDAL) or Java Topology Suite (JTS) in ad hoc user-defined functions (UDFs) for processing in a distributed fashion with Spark DataFrames. This is the simplest approach for scaling existing workloads without much code rewrite; however, it can introduce performance drawbacks as it is more lift-and-shift in nature.

3. Indexing the data with grid systems and leveraging the generated index to perform spatial operations is a common approach for dealing with very large-scale or computationally restricted workloads. S2, GeoHex and Uber's H3 are examples of such grid systems. Grids approximate geo features such as polygons or points with a fixed set of identifiable cells, thus avoiding expensive geospatial operations altogether, and thus offer much better scaling behavior. Implementers can decide between grids fixed to a single accuracy that can be somewhat lossy yet more performant or grids with multiple accuracies that can be less performant but mitigate against lossines.

The following examples are generally oriented around a New York City taxi pickup/drop-off data set found here. NYC Taxi Zone data with geometries will also be used as the set of polygons. This data contains polygons for the five boroughs of NYC as well the neighborhoods. This notebook will walk you through preparations and cleanings done to convert the initial CSV files into Delta Lake tables as a reliable and performant data source.

Our base DataFrame is the taxi pickup/drop-off data read from a Delta Lake Table using Databricks.

```
%scala
val dfRaw = spark.read.format("delta").load("/ml/blogs/geospatial/
delta/nyc-green")
display(dfRaw) // showing first 10 columns
```



vendor_id	pickup_datetime	dropoff_datetime	store_and_forward	rate_code_id	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passener_count
2	2017-09-30 23:48:04	2017-09-30 23:57:43	N	1	82	7	2	1.89	9
2	2017-09-30 23:50:24	2017-09-30 23:55:30	N	1	25	181	6	1.26	6
2	2017-09-30 23:28:29	2017-09-30 23:37:29	N	1	41	159	1	2.28	9
2	2017-09-30 23:46:44	2017-09-30 23:54:59	N	1	42	41	1	1.09	7
2	2017-09-30	2017-09-30 23:31:49	N	1	33	189	1	2.35	10

Showing the first 1000 rows.

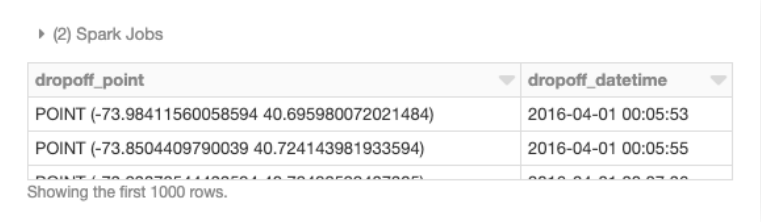
Figure 1: Geospatial data read from a Delta Lake table using Databricks

Geospatial operations using geospatial libraries for Apache Spark

Over the last few years, several libraries have been developed to extend the capabilities of Apache Spark for geospatial analysis. These frameworks bear the brunt of registering commonly applied user-defined types (UDT) and functions (UDF) in a consistent manner, lifting the burden otherwise placed on users and teams to write ad hoc spatial logic. Please note that in this blog post, we use several different spatial frameworks chosen to highlight various capabilities. We understand that other frameworks exist beyond those highlighted, which you might also want to use with Databricks to process your spatial workloads.

Earlier, we loaded our base data into a DataFrame. Now we need to turn the latitude/longitude attributes into point geometries. To accomplish this, we will use UDFs to perform operations on DataFrames in a distributed fashion. Please refer to the provided notebooks at the end of the blog for details on adding these frameworks to a cluster and the initialization calls to register UDFs and UDTs. For starters, we have added GeoMesa to our cluster, a framework especially adept at handling vector data. For ingestion, we are mainly leveraging its integration of JTS with Spark SQL, which allows us to easily convert to and use registered JTS geometry classes. We will be using the function st_makePoint that, given a latitude and longitude, create a Point geometry object. Since the function is a UDF, we can apply it to columns directly.

```
%scala
val df = dfRaw
 .withColumn("pickup_point", st_makePoint(col("pickup_longitude"),
col("pickup_latitude")))
 .withColumn("dropoff_point", st_makePoint(col("dropoff_
longitude"),col("dropoff_latitude")))
display(df.select("dropoff_point","dropoff_datetime"))
```



▸ (2) Spark Jobs

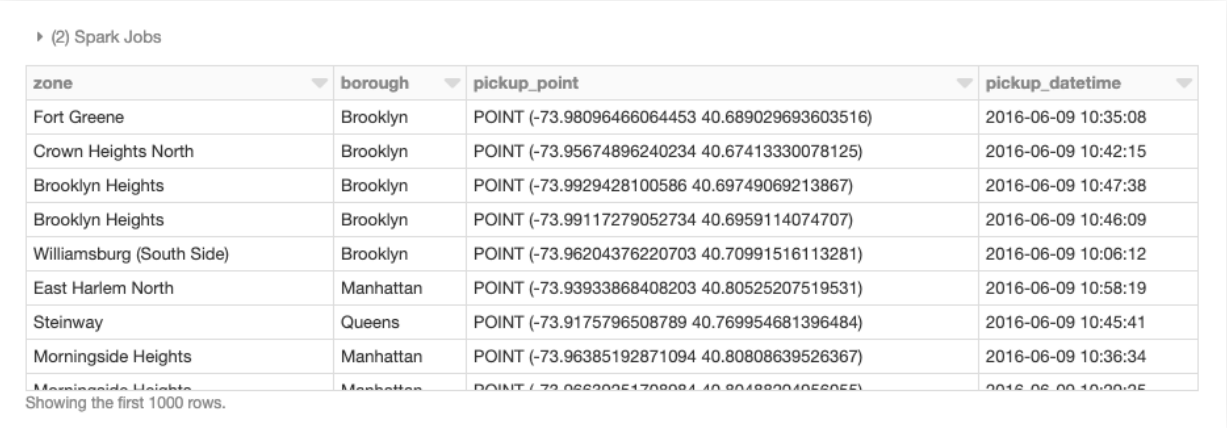
dropoff_point	dropoff_datetime
POINT (-73.98411560058594 40.695980072021484)	2016-04-01 00:05:53
POINT (-73.8504409790039 40.724143981933594)	2016-04-01 00:05:55

Showing the first 1000 rows.

Figure 2: Using UDFs to perform operations on DataFrames in a distributed fashion to turn geospatial data latitude/longitude attributes into point geometries.

We can also perform distributed spatial joins, in this case using GeoMesa's provided st_contains UDF to produce the resulting join of all polygons against pickup points.

```
%scala
val joinedDF = wktDF.join(df, st_contains($”the_geom”, $”pickup_point”)
display(joinedDF.select(“zone”,”borough”,”pickup_point”,”pickup_
datetime”))
```



▸ (2) Spark Jobs

zone	borough	pickup_point	pickup_datetime
Fort Greene	Brooklyn	POINT (-73.98096466064453 40.689029693603516)	2016-06-09 10:35:08
Crown Heights North	Brooklyn	POINT (-73.95674896240234 40.67413330078125)	2016-06-09 10:42:15
Brooklyn Heights	Brooklyn	POINT (-73.9929428100586 40.69749069213867)	2016-06-09 10:47:38
Brooklyn Heights	Brooklyn	POINT (-73.99117279052734 40.6959114074707)	2016-06-09 10:46:09
Williamsburg (South Side)	Brooklyn	POINT (-73.96204376220703 40.70991516113281)	2016-06-09 10:06:12
East Harlem North	Manhattan	POINT (-73.93933868408203 40.80525207519531)	2016-06-09 10:58:19
Steinway	Queens	POINT (-73.9175796508789 40.769954681396484)	2016-06-09 10:45:41
Morningside Heights	Manhattan	POINT (-73.96385192871094 40.80808639526367)	2016-06-09 10:36:34

Showing the first 1000 rows.

Figure 3: Using GeoMesa's provided st_contains UDF, for example, to produce the resulting join of all polygons against pickup points

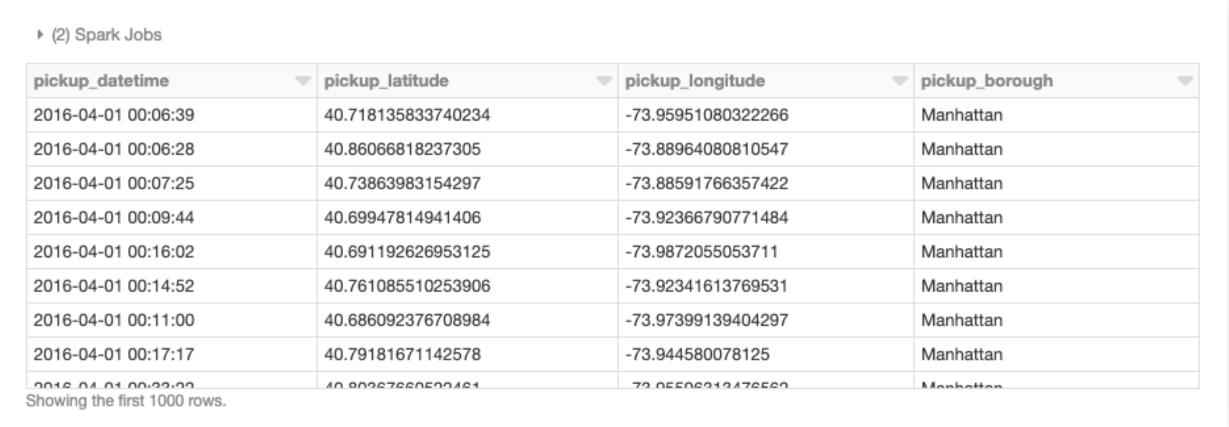
Wrapping single-node libraries in UDFs

In addition to using purpose-built distributed spatial frameworks, existing single-node libraries can also be wrapped in ad hoc UDFs for performing geospatial operations on DataFrames in a distributed fashion. This pattern is available to all Spark language bindings — Scala, Java, Python, R and SQL — and is a simple approach for leveraging existing workloads with minimal code changes. To demonstrate a single-node example, let's load NYC borough data and define UDF find_borough(…) for point-in-polygon operation to assign each GPS location to a borough using geopandas. This could also have been accomplished with a vectorized UDF for even better performance

```
%python
# read the boroughs polygons with geopandas
gdf = gdp.read_file(“/dbfs/ml/blogs/geospatial/nyc_boroughs.geojson”)

b_gdf = sc.broadcast(gdf) # broadcast the geopandas dataframe to all
nodes of the cluster
def find_borough(latitude,longitude):
  mgdf = b_gdf.value.apply(lambda x: x[“boro_name”] if x[“geometry”].
intersects(Point(longitude, latitude))
  idx = mgdf.first_valid_index()
  return mgdf.loc[idx] if idx is not None else None

find_borough_udf = udf(find_borough, StringType())
```

Now we can apply the UDF to add a column to our Spark DataFrame, which assigns a borough name to each pickup point.

```
%python
# read the coordinates from delta
df = spark.read.format(“delta”).load(“/ml/blogs/geospatial/delta/nyc-
green”)
df_with_boroughs = df.withColumn(“pickup_borough”, find_borough_
udf(col(“pickup_latitude”),col(pickup_longitude)))
display(df_with_boroughs.select(
  “pickup_datetime”,”pickup_latitude”,”pickup_longitude”,”pickup_
borough”))
```



▸ (2) Spark Jobs

pickup_datetime	pickup_latitude	pickup_longitude	pickup_borough
2016-04-01 00:06:39	40.718135833740234	-73.95951080322266	Manhattan
2016-04-01 00:06:28	40.86066818237305	-73.88964080810547	Manhattan
2016-04-01 00:07:25	40.73863983154297	-73.88591766357422	Manhattan
2016-04-01 00:09:44	40.69947814941406	-73.92366790771484	Manhattan
2016-04-01 00:16:02	40.691192626953125	-73.9872055053711	Manhattan
2016-04-01 00:14:52	40.761085510253906	-73.92341613769531	Manhattan
2016-04-01 00:11:00	40.686092376708984	-73.97399139404297	Manhattan
2016-04-01 00:17:17	40.79181671142578	-73.944580078125	Manhattan
2016-04-01 00:23:22	40.80267660522461	-73.95596313476562	Manhattan

Showing the first 1000 rows.

Figure 4: The result of a single-node example, where GeoPandas is used to assign each GPS location to an NYC borough

Grid systems for spatial indexing

Geospatial operations are inherently computationally expensive. Point-in-polygon, spatial joins, nearest neighbor or snapping to routes all involve complex operations. By indexing with grid systems, the aim is to avoid geospatial operations altogether. This approach leads to the most scalable implementations with the caveat of approximate operations. Here is a brief example with H3.

Scaling spatial operations with H3 is essentially a two-step process. The first step is to compute an H3 index for each feature (points, polygons, …) defined as UDF geoToH3(…). The second step is to use these indices for spatial operations such as spatial join (point-in-polygon, k-nearest neighbors, etc.), in this case defined as UDF multiPolygonToH3(…).

```
%scala
import com.uber.h3core.H3Core
import com.uber.h3core.util.GeoCoord
import scala.collection.JavaConversions._
import scala.collection.JavaConverters._

object H3 extends Serializable {
  val instance = H3Core.newInstance()
}

val geoToH3 = udf{ (latitude: Double, longitude: Double, resolution:
Int) =>
  H3.instance.geoToH3(latitude, longitude, resolution)
}

val polygonToH3 = udf{ (geometry: Geometry, resolution: Int) =>
  var points: List[GeoCoord] = List()
  var holes: List[java.util.List[GeoCoord]] = List()
  if (geometry.getGeometryType == “Polygon”) {
    points = List(
      geometry
        .getCoordinates()
        .toList
        .map(coord => new GeoCoord(coord.y, coord.x)): _*)
  }
  H3.instance.polyfill(points, holes.asJava, resolution).toList
}
```

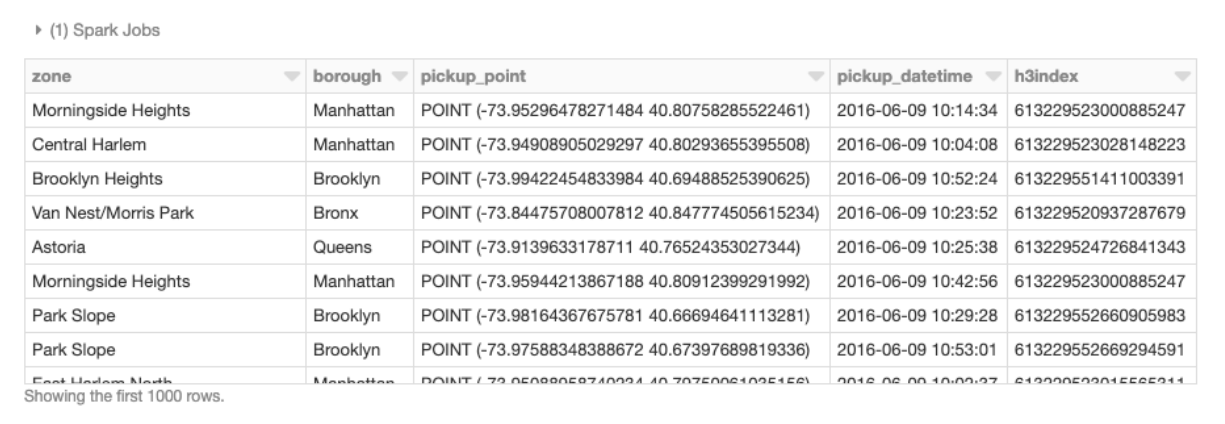
```
val multiPolygonToH3 = udf{ (geometry: Geometry, resolution: Int) =>
  var points: List[GeoCoord] = List()
  var holes: List[java.util.List[GeoCoord]] = List()
  if (geometry.getGeometryType == “MultiPolygon”) {
    val numGeometries = geometry.getNumGeometries()
    if (numGeometries > 0) {
      points = List(
        geometry
          .getGeometryN(0)
          .getCoordinates()
          .toList
          .map(coord => new GeoCoord(coord.y, coord.x)): _*)
    }
    if (numGeometries > 1) {
      holes = (1 to (numGeometries - 1)).toList.map(n => {
        List(
          geometry
            .getGeometryN(n)
            .getCoordinates()
            .toList
            .map(coord => new GeoCoord(coord.y, coord.x)): _*).asJava
      })
    }
  }
  H3.instance.polyfill(points, holes.asJava, resolution).toList
}
```

We can now apply these two UDFs to the NYC taxi data as well as the set of borough polygons to generate the H3 index.

```
%scala
val res = 7 //the resolution of the H3 index, 1.2km
val dfH3 = df.withColumn(
  “h3index”,
  geoToH3(col(“pickup_latitude”), col(“pickup_longitude”), lit(res))
)
val wktDFH3 = wktDF
  .withColumn(“h3index”, multiPolygonToH3(col(“the_geom”), lit(res)))
  .withColumn(“h3index”, explode($”h3index”))
```

Given a set of lat/lon points and a set of polygon geometries, it is now possible to perform the spatial join using h3index field as the join condition. These assignments can be used to aggregate the number of points that fall within each polygon, for instance. There are usually millions or billions of points that have to be matched to thousands or millions of polygons, which necessitates a scalable approach. There are other techniques not covered in this blog that can be used for indexing in support of spatial operations when an approximation is insufficient.

```
%scala
val dfWithBoroughH3 = dfH3.join(wktDFH3,”h3index”)

display(df_with_borough_h3.select(“zone”,”borough”,”pickup_
point”,”pickup_datetime”,”h3index”))
```



(1) Spark Jobs

zone	borough	pickup_point	pickup_datetime	h3index
Morningside Heights	Manhattan	POINT (-73.95296478271484 40.80758285522461)	2016-06-09 10:14:34	613229523000885247
Central Harlem	Manhattan	POINT (-73.94908905029297 40.80293655395508)	2016-06-09 10:04:08	613229523028148223
Brooklyn Heights	Brooklyn	POINT (-73.99422454833984 40.69488525390625)	2016-06-09 10:52:24	613229551411003391
Van Nest/Morris Park	Bronx	POINT (-73.84475708007812 40.847774505615234)	2016-06-09 10:23:52	613229520937287679
Astoria	Queens	POINT (-73.9139633178711 40.76524353027344)	2016-06-09 10:25:38	613229524726841343
Morningside Heights	Manhattan	POINT (-73.95944213867188 40.80912399291992)	2016-06-09 10:42:56	613229523000885247
Park Slope	Brooklyn	POINT (-73.98164367675781 40.66694641113281)	2016-06-09 10:29:28	613229552660905983
Park Slope	Brooklyn	POINT (-73.97588348388672 40.67397689819336)	2016-06-09 10:53:01	613229552669294591
East Harlem North	Manhattan	[illegible]	[illegible]	[illegible]

Showing the first 1000 rows.

Figure 5: DataFrame table representing the spatial join of a set of lat/lon points and polygon geometries, using a specific field as the join condition

Here is a visualization of taxi drop-off locations, with latitude and longitude binned at a resolution of 7 (1.22km edge length) and colored by aggregated counts within each bin.

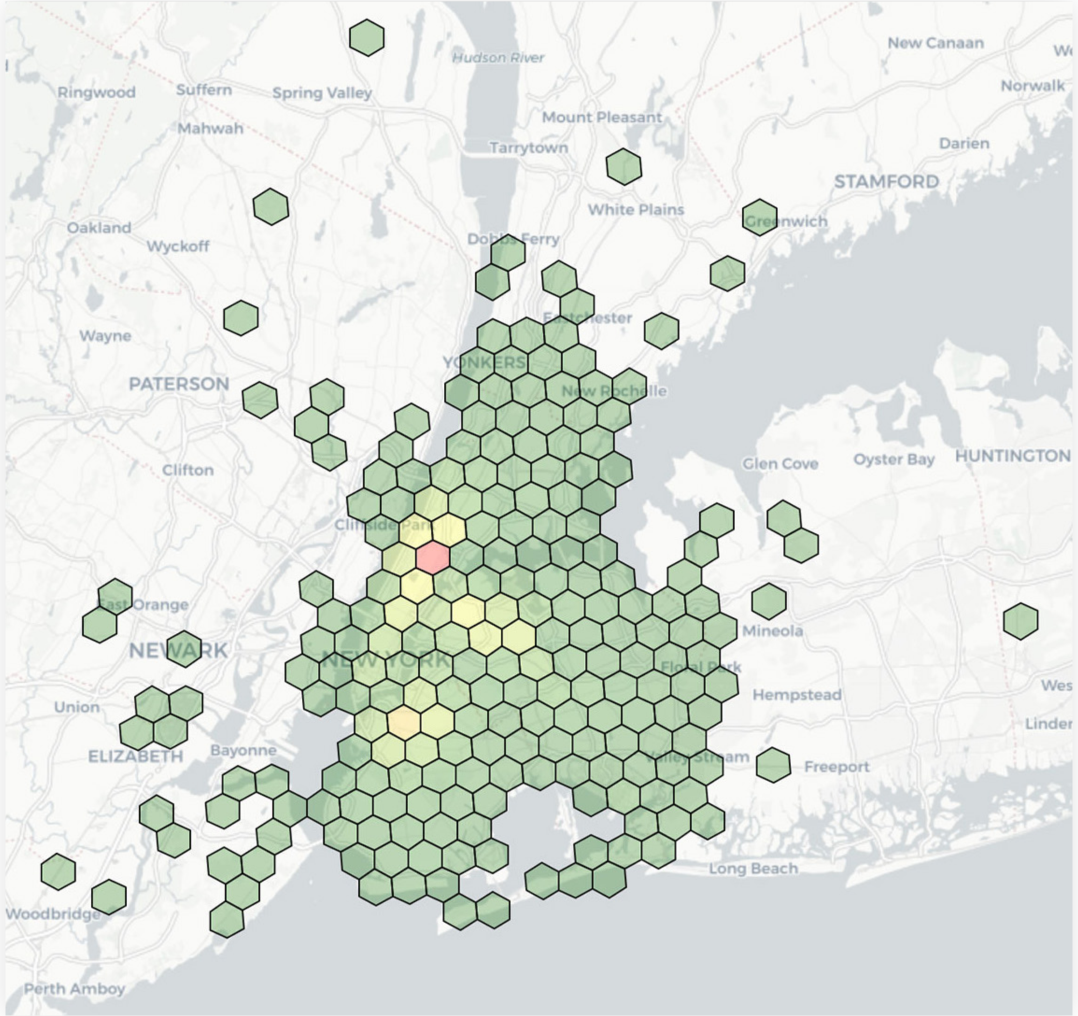


Figure 6: Geospatial visualization of taxi drop-off locations, with latitude and longitude binned at a resolution of 7 (1.22km edge length) and colored by aggregated counts within each bin

Handling spatial formats with Databricks

Geospatial data involves reference points, such as latitude and longitude, to physical locations or extents on the Earth along with features described by attributes. While there are many file formats to choose from, we have picked out a handful of representative vector and raster formats to demonstrate reading with Databricks.

Vector data

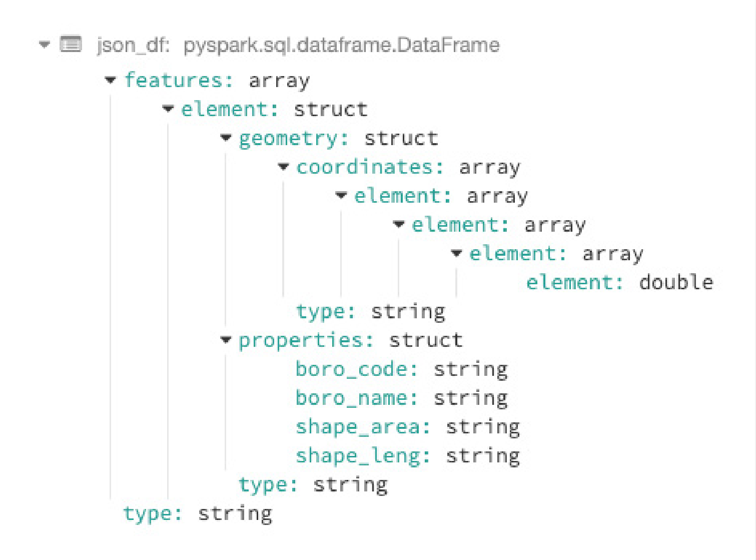
Vector data is a representation of the world stored in x (longitude), y (latitude) coordinates in degrees, and also z (altitude in meters) if elevation is considered. The three basic symbol types for vector data are points, lines and polygons. Well-known-text (WKT), GeoJSON and shapefile are some popular formats for storing vector data we highlight below.

Let's read NYC Taxi Zone data with geometries stored as WKT. The data structure we want to get back is a DataFrame that will allow us to standardize with other APIs and available data sources, such as those used elsewhere in the blog. We are able to easily convert the WKT content found in field the_geom into its corresponding JTS Geometry class through the st_geomFromWKT(...) UDF call.

```
%scala
val wktDFText = sqlContext.read.format(“csv”)
 .option(“header”, “true”)
 .option(“inferSchema”, “true”)
 .load(“/ml/blogs/geospatial/nyc_taxi_zones.wkt.csv”)

val wktDF = wktDFText.withColumn(“the_geom”, st_geomFromWKT(col(“the_
geom”))).cache
```

GeoJSON is used by many open source GIS packages for encoding a variety of geographic data structures, including their features, properties and spatial extents. For this example, we will read NYC Borough Boundaries with the approach taken depending on the workflow. Since the data is conforming to JSON, we could use the Databricks built-in JSON reader with .option("multiline","true") to load the data with the nested schema.

```
%python
json_df = spark.read.option("multiline","true").json("nyc_boroughs.
geojson")
```

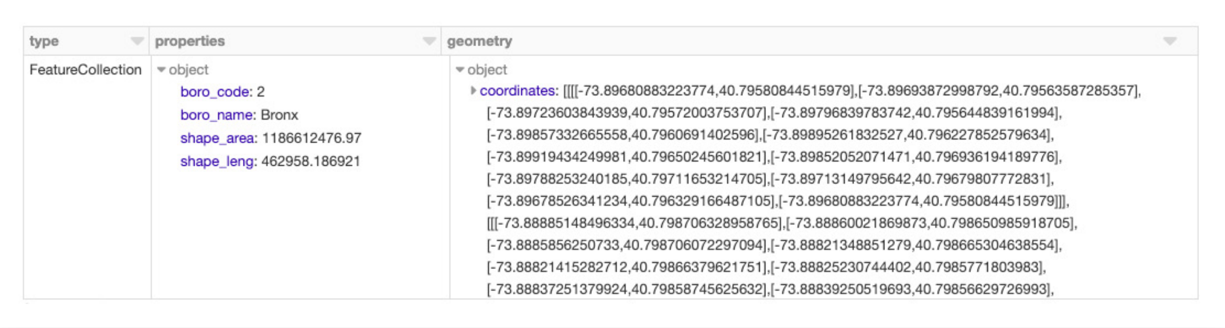


```
▾ json_df:  pyspark.sql.dataframe.DataFrame
    ▾ features: array
        ▾ element: struct
            ▾ geometry: struct
                ▾ coordinates: array
                    ▾ element: array
                        ▾ element: array
                            ▾ element: array
                                  element: double
                  type: string
            ▾ properties: struct
                  boro_code: string
                  boro_name: string
                  shape_area: string
                  shape_leng: string
              type: string
      type: string
```

Figure 7: Using the Databricks built-in JSON reader .option("multiline","true") to load the data with the nested schema

From there, we could choose to hoist any of the fields up to top level columns using Spark's built-in explode function. For example, we might want to bring up geometry, properties and type and then convert geometry to its corresponding JTS class, as was shown with the WKT example.

```
%python
from pyspark.sql import functions as F
json_explode_df = ( json_df.select(
 "features",
 "type",
 F.explode(F.col("features.properties")).alias("properties")
).select("*",F.explode(F.col("features.geometry")).alias("geometry")).
drop("features"))

display(json_explode_df)
```



type	properties	geometry
FeatureCollection	▾ object boro_code: 2 boro_name: Bronx shape_area: 1186612476.97 shape_leng: 462958.186921	▾ object ▸ coordinates: [[[[-73.89680883223774,40.79580844515979],[-73.89693872998792,40.7956358728 5357], [-73.89723603843939,40.79572003753707],[-73.89796839783742,40.795644839161994], [-73.89857332665558,40.7960691402596],[-73.89895261832527,40.79622785257963 4], [-73.89919434249981,40.79650245601821],[-73.89852052071471,40.796936194189776], [-73.89788253240185,40.79711653214705],[-73.89713149795642,40.79679807772831], [-73.89678526341234,40.796329166487105],[-73.89680883223774,40.79580844515979]]], [[[-73.88885148496334,40.798706328958765],[-73.88860021869873,40.798650985918705], [-73.8885856250733,40.798706072297094],[-73.88821348851279,40.798665304638554], [-73.88821415282712,40.79866379621751],[-73.88825230744402,40.7985771803983], [-73.88837251379924,40.79858745625632],[-73.88839250519693,40.79856629726993],

Figure 8: Using the Spark's built-in explode function to raise a field to the top level, displayed within a DataFrame table

We can also visualize the NYC Taxi Zone data within a notebook using an existing DataFrame or directly rendering the data with a library such as Folium, a Python library for rendering spatial data. Databricks File System (DBFS) runs over a distributed storage layer, which allows code to work with data formats using familiar file system standards. DBFS has a FUSE Mount to allow local API calls that perform file read and write operations, which makes it very easy to load data with non-distributed APIs for interactive rendering. In the Python open(…) command below, the "/dbfs/…" prefix enables the use of FUSE Mount.

```
%python
import folium
import json

with open ("/dbfs/ml/blogs/geospatial/nyc_boroughs.geojson", "r") as
myfile:
 boro_data=myfile.read() # read GeoJSON from DBFS using FuseMount

m = folium.Map(
 location=[40.7128, -74.0060],
 tiles='Stamen Terrain',
 zoom_start=12
)
folium.GeoJson(json.loads(boro_data)).add_to(m)
m # to display, also could use displayHTML(...) variants
```

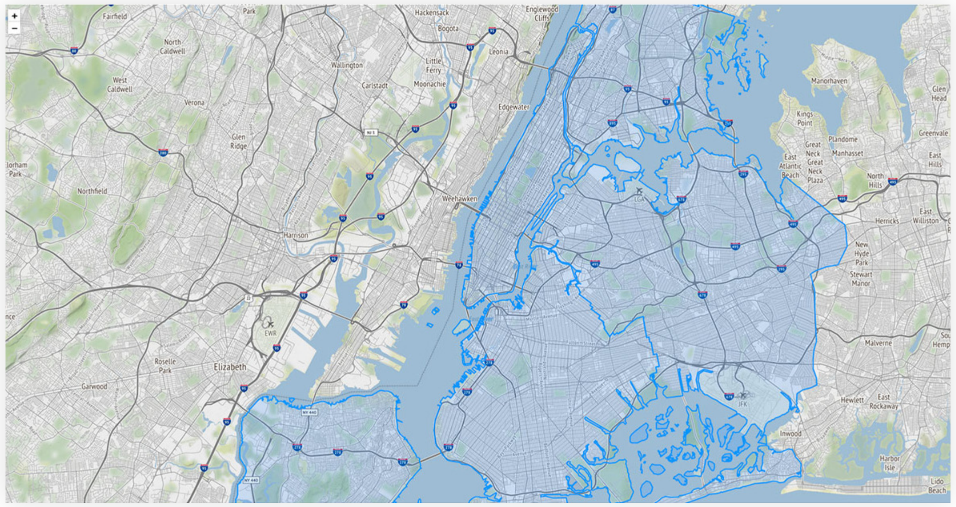


Figure 9: We can also visualize the NYC Taxi Zone data, for example, within a notebook using an existing DataFrame or directly rendering the data with a library such as Folium, a Python library for rendering geospatial data

Shapefile is a popular vector format developed by ESRI that stores the geometric location and attribute information of geographic features. The format consists of a collection of files with a common filename prefix (*.shp, *.shx and *.dbf are mandatory) stored in the same directory. An alternative to shapefile is KML, also used by our customers but not shown for brevity. For this example, let's use NYC Building shapefiles. While there are many ways to demonstrate reading shapefiles, we will give an example using GeoSpark. The built-in ShapefileReader is used to generate the rawSpatialDf DataFrame.

```
%scala
var spatialRDD = new SpatialRDD[Geometry]
spatialRDD = ShapefileReader.readToGeometryRDD(sc, "/ml/blogs/
geospatial/shapefiles/nyc")

var rawSpatialDf = Adapter.toDf(spatialRDD,spark)
rawSpatialDf.createOrReplaceTempView("rawSpatialDf") //DataFrame now
available to SQL, Python, and R
```

By registering rawSpatialDf as a temp view, we can easily drop into pure Spark SQL syntax to work with the DataFrame, to include applying a UDF to convert the shapefile WKT into Geometry.

```
%sql
SELECT *,
 ST_GeomFromWKT(geometry) AS geometry -- GeoSpark UDF to convert WKT to
Geometry
FROM rawspatialdf
```

Additionally, we can use Databricks' built-in visualization for in-line analytics, such as charting the tallest buildings in NYC.

```
%sql
SELECT name,
 round(Cast(num_floors AS DOUBLE), 0) AS num_floors --String to Number
FROM rawspatialdf
WHERE name  ‘’
ORDER BY num_floors DESC LIMIT 5
```

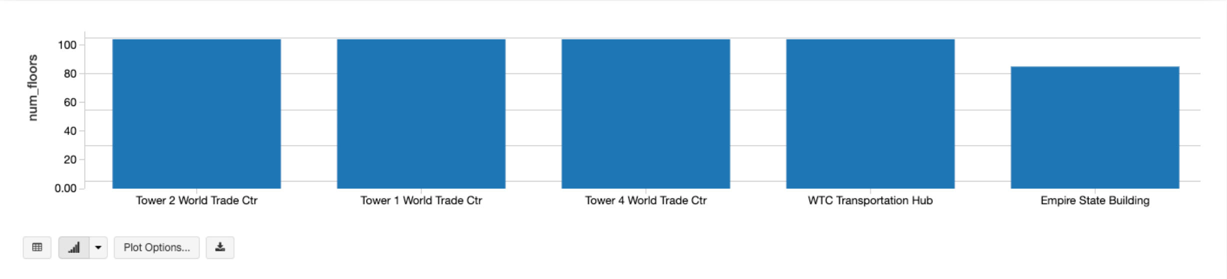


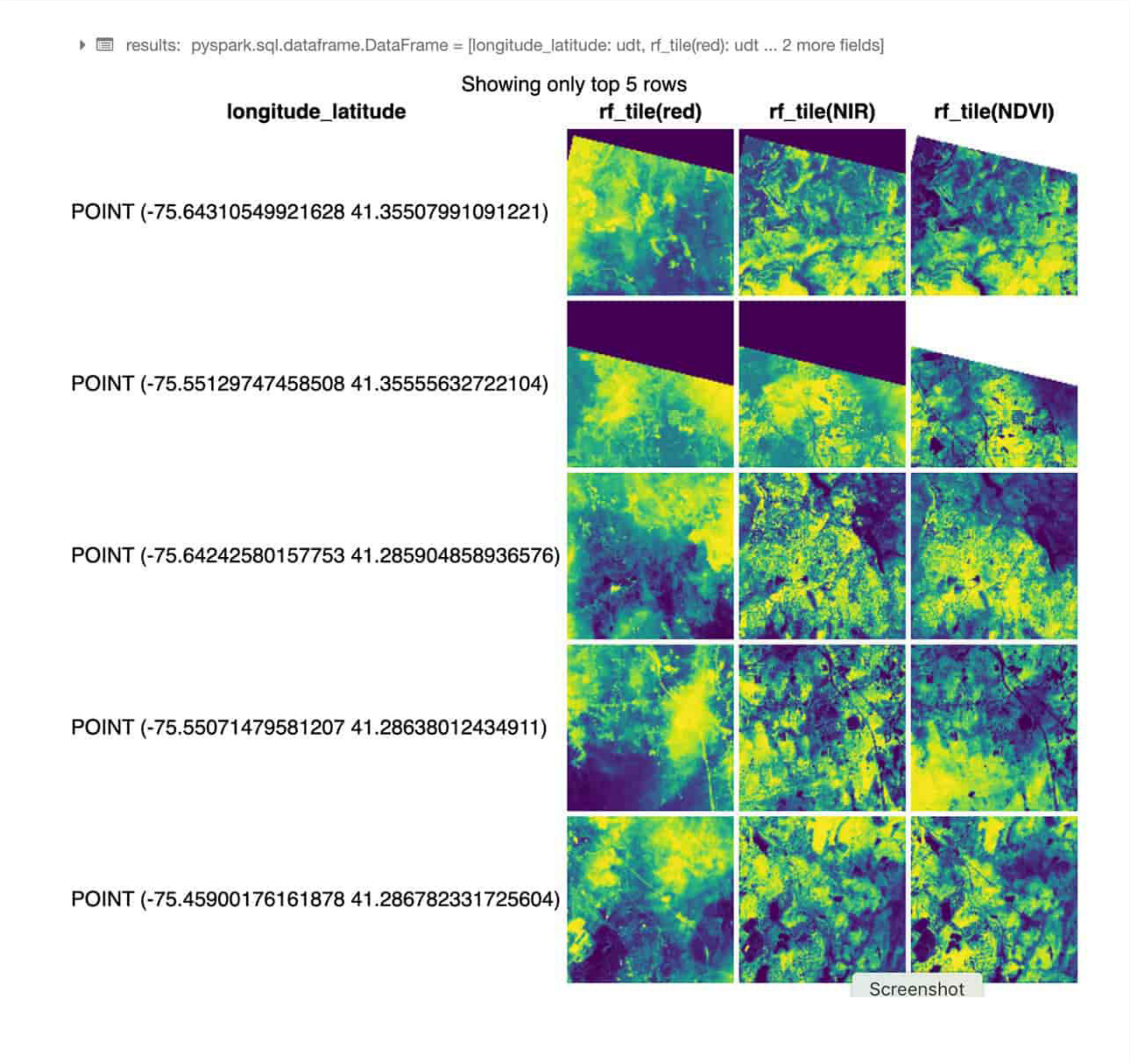
Figure 10: A Databricks built-in visualization for in-line analytics charting, for example, the tallest buildings in NYC

Raster data

Raster data stores information of features in a matrix of cells (or pixels) organized into rows and columns (either discrete or continuous). Satellite images, photogrammetry and scanned maps are all types of raster-based Earth Observation (EO) data.

The following Python example uses RasterFrames, a DataFrame-centric spatial analytics framework, to read two bands of GeoTIFF Landsat-8 imagery (red and near-infrared) and combine them into Normalized Difference Vegetation Index. We can use this data to assess plant health around NYC. The rf_ipython module is used to manipulate RasterFrame contents into a variety of visually useful forms, such as below where the red, NIR and NDVI tile columns are rendered with color ramps, using the Databricks built-in displayHTML(…) command to show the results within the notebook.

```
%python
# construct a CSV “catalog” for RasterFrames `raster` reader
# catalogs can also be Spark or
```



```
results:  pyspark.sql.dataframe.DataFrame = [longitude_latitude: udt, rf_tile(red): udt ... 2 more fields]
```

Showing only top 5 rows

longitude_latitude	rf_tile(red)	rf_tile(NIR)	rf_tile(NDVI)
POINT (-75.64310549921628 41.35507991091221)			
POINT (-75.55129747458508 41.35555632722104)			
POINT (-75.64242580157753 41.285904858936576)			
POINT (-75.55071479581207 41.28638012434911)			
POINT (-75.45900176161878 41.286782331725604)			

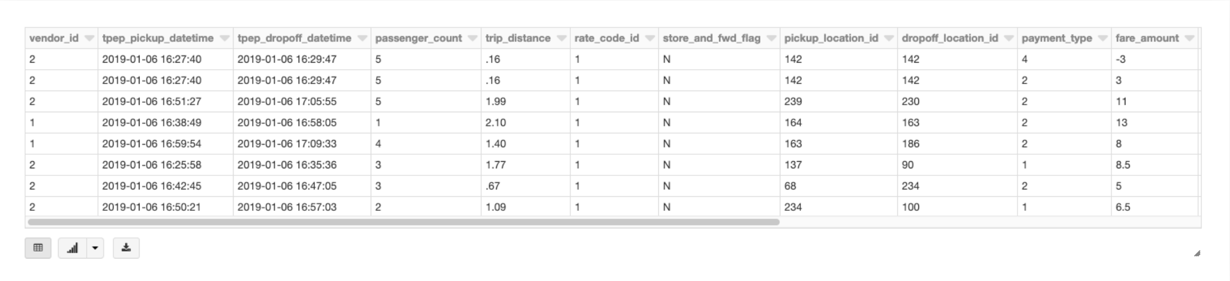
Figure 11: RasterFrame contents can be filtered, transformed, summarized, resampled and rasterized through over 200 raster and vector functions

Through its custom Spark DataSource, RasterFrames can read various raster formats, including GeoTIFF, JP2000, MRF and HDF, from an array of services. It also supports reading the vector formats GeoJSON and WKT/WKB. RasterFrame contents can be filtered, transformed, summarized, resampled and rasterized through over 200 raster and vector functions, such as st_reproject(...) and st_centroid(...) used in the example above. It provides APIs for Python, SQL and Scala as well as interoperability with Spark ML.

Geodatabases

Geodatabases can be file based for smaller scale data or accessible via JDBC/ODBC connections for medium scale data. You can use Databricks to query many SQL databases with the built-in JDBC/ODBC Data Source. Connecting to PostgreSQL is shown below and is commonly used for smaller scale workloads by applying PostGIS extensions. This pattern of connectivity allows customers to maintain as-is access to existing databases.

```
%scala
display(
  sqlContext.read.format("jdbc")
    .option("url", jdbcUrl)
    .option("driver", "org.postgresql.Driver")
    .option("dbtable",
      """(SELECT * FROM yellow_tripdata_staging
      OFFSET 5 LIMIT 10) AS t""") //predicate pushdown
    .option("user", jdbcUsername)
    .option("jdbcPassword", jdbcPassword)
  .load)
```



vendor_id	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	rate_code_id	store_and_fwd_flag	pickup_location_id	dropoff_location_id	payment_type	fare_amount
2	2019-01-06 16:27:40	2019-01-06 16:29:47	5	.16	1	N	142	142	4	-3
2	2019-01-06 16:27:40	2019-01-06 16:29:47	5	.16	1	N	142	142	2	3
2	2019-01-06 16:51:27	2019-01-06 17:05:55	5	1.99	1	N	239	230	2	11
1	2019-01-06 16:38:49	2019-01-06 16:58:05	1	2.10	1	N	164	163	2	13
1	2019-01-06 16:59:54	2019-01-06 17:09:33	4	1.40	1	N	163	186	2	8
2	2019-01-06 16:25:58	2019-01-06 16:35:36	3	1.77	1	N	137	90	1	8.5
2	2019-01-06 16:42:45	2019-01-06 16:47:05	3	.67	1	N	68	234	2	5
2	2019-01-06 16:50:21	2019-01-06 16:57:03	2	1.09	1	N	234	100	1	6.5

Getting started with geospatial analysis on Databricks

Businesses and government agencies seek to use spatially referenced data in conjunction with enterprise data sources to draw actionable insights and deliver on a broad range of innovative use cases. In this blog we demonstrated how the Databricks Unified Data Analytics Platform can easily scale geospatial workloads, enabling our customers to harness the power of the cloud to capture, store and analyze data of massive size.

In an upcoming blog, we will take a deep dive into more advanced topics for geospatial processing at-scale with Databricks. You will find additional details about the spatial formats and highlighted frameworks by reviewing Data Prep Notebook, GeoMesa + H3 Notebook, GeoSpark Notebook, GeoPandas Notebook, and RasterFrames Notebook. Also, stay tuned for a new section in our documentation specifically for geospatial topics of interest.

CHAPTER 9:

Exploring Twitter Sentiment and Crypto Price Correlation Using Databricks

By **Monica Lin**, **Christoph Meier**, **Matthew Parker** and **Kiran Ravella**

Introduction

The market capitalization of cryptocurrencies increased from $17 billion in 2017 to $2.25 trillion in 2021. That's over a 13,000% ROI in a short span of 5 years! Even with this growth, cryptocurrencies remain incredibly volatile, with their value being impacted by a multitude of factors: market trends, politics, technology … and Twitter. Yes, that's right. There have been instances where their prices were impacted on account of tweets by famous personalities.

As part of a data engineering and analytics course at the Harvard Extension School, our group worked on a project to create a cryptocurrency data lake for different data personas — including data engineers, ML practitioners and BI analysts — to analyze trends over time, particularly the impact of social media on the price volatility of a crypto asset, such as Bitcoin (BTC). We leveraged the Databricks Lakehouse Platform to ingest unstructured data from Twitter using the Tweepy library and traditional structured pricing data from Yahoo Finance to create a machine learning prediction model that analyzes the impact of investor sentiment on crypto asset valuation. The aggregated trends and actionable insights are presented on a Databricks SQL dashboard, allowing for easy consumption to relevant stakeholders.

This blog walks through how we built this ML model in just a few weeks by leveraging Databricks and its collaborative notebooks. We would like to thank the Databricks University Alliance program and the extended team for all the support.

Overview

One advantage of cryptocurrency for investors is that it is traded 24/7 and the market data is available round the clock. This makes it easier to analyze the correlation between the Tweets and crypto prices. A high-level architecture of the data and ML pipeline is presented in figure 1 below.

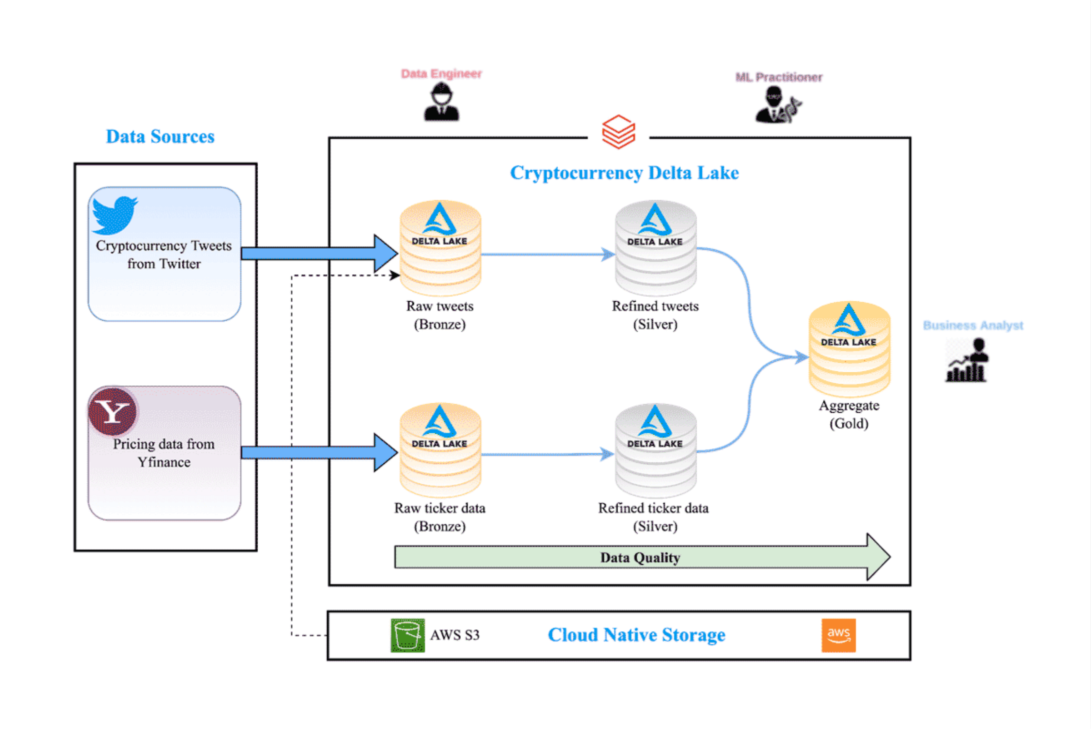


Figure 1: Crypto Lake using Delta

The full orchestration workflow runs a sequence of Databricks notebooks that perform the following tasks:

1. Data ingestion pipeline

Imports the raw data into the Cryptocurrency Delta Lake Bronze tables

2. Data science

- Cleans data and applies the Twitter sentiment machine learning model into Silver tables
- Aggregates the refined Twitter and Yahoo Finance data into an aggregated Gold table
- Computes the correlation ML model between price and sentiment

3. Data analysis

Runs updated SQL BI queries on the Gold table

The lakehouse paradigm combines key capabilities of data lakes and data warehouses to enable all kinds of BI and AI use cases. The use of the lakehouse architecture enabled rapid acceleration of the pipeline creation to just one week. As a team, we played specific roles to mimic different data personas, and this paradigm facilitated the seamless handoffs between data engineering, machine learning and business intelligence roles without requiring data to be moved across systems.

Data/ML pipeline

Ingestion using a medallion architecture

The two primary data sources were Twitter and Yahoo Finance. A lookup table was used to hold the crypto tickers and their Twitter hashtags to facilitate the subsequent search for associated tweets.

We used yfinance python library to download historical crypto exchange market data from Yahoo Finance's API in 15-minute intervals. The raw data was stored in a Bronze table containing information such as ticker symbol, datetime, open, close, high, low and volume. We then created a Delta Lake Silver table with additional data, such as the relative change in price of the ticker in that interval. Using Delta Lake made it easy to reprocess the data, as it guarantees atomicity with every operation. It also ensures that schema is enforced and prevents bad data from creeping into the lake.

We used Tweepy Python library to download Twitter data. We stored the raw tweets in a Delta Lake Bronze table. We removed unnecessary data from the Bronze table and also filtered out non-ASCII characters like emojis. This refined data was stored in a Delta Lake Silver table.

Data science

The data science portion of our project consists of three major parts: exploratory data analysis, sentiment model and correlation model. The objective is to build a sentiment model and use the output of the model to evaluate the correlation between sentiment and the prices of different cryptocurrencies, such as Bitcoin, Ethereum, Coinbase and Binance. In our case, the sentiment model follows a supervised, multi-class classification approach, while the correlation model uses a linear regression model. Lastly, we used MLflow for both models' lifecycle management, including experimentation, reproducibility, deployment and a central model registry. MLflow Model Registry collaboratively manages the full lifecycle of an MLflow model by offering a centralized model store, set of APIs and UI. Some of its most useful features include model lineage (which MLflow experiment and run produced the model), model versioning, stage transitions (such as from staging to production or archiving), and annotations.

Exploratory data analysis

The EDA section provides insightful visualizations on the data set. For example, we looked at the distribution of tweet lengths for each sentiment category using violin plots from Seaborn. Word clouds (using Matplotlib and wordcloud libraries) for positive and negative tweets were also used to show the most common words for the two sentiment types. Lastly, an interactive topic modeling dashboard was built, using Gensim, to provide insights on the top most common topics in the data set and the most frequently used words in each topic, as well as how similar the topics are to each other.

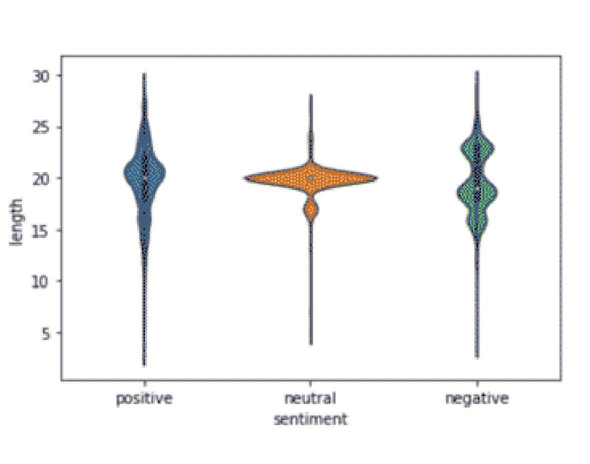


Figure 2: Violin plots for text length

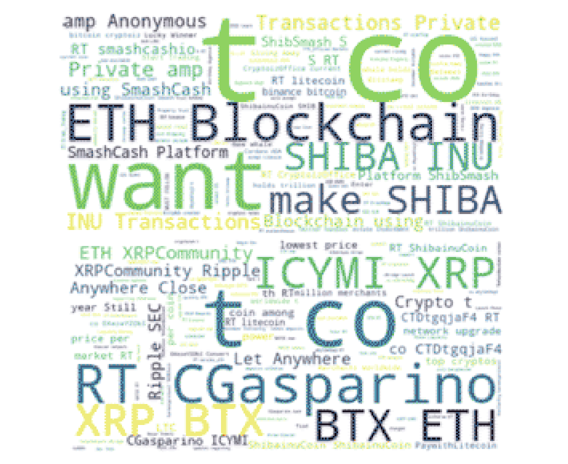
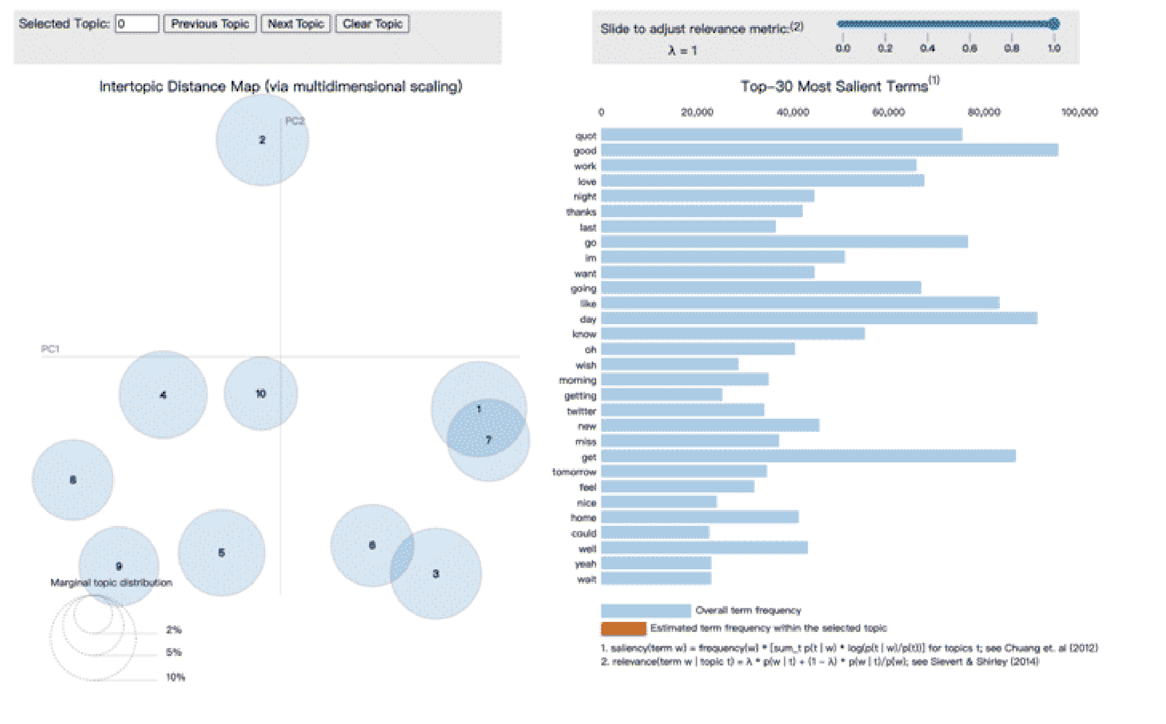


Figure 3: Word cloud for positive and negative tweets



1. saliency(term w) = frequency(w) * [sum_t p(t | w) * log(p(t | w)/p(t))] for topics t; see Chuang et. al (2012)
2. relevance(term w | topic t) = λ * p(w | t) + (1 - λ) * p(w | t)/p(w); see Sievert & Shirley (2014)

Figure 4: Interactive topic modeling dashboard

Sentiment analysis model

Developing a proper sentiment analysis model has been one of the core tasks within the project. In our case, the goal of this model was to classify the polarities that are expressed in raw tweets as input using a mere polar view of sentiment (i.e., tweets were categorized as "positive," "neutral" or "negative"). Since sentiment analysis is a problem of great practical relevance, it is no surprise that multiple ML strategies related to it can be found in literature:

Sentiment lexicons algorithms	Off-the-shelf sentiment analysis systems
Compare each word in a tweet to a database of words that are labeled as having positive or negative sentiment A tweet with more positive words than negative would be scored as a positive and vice versa Pros: straightforward approach Cons: performs poorly in general and greatly depends on the quality of the database of words	Exemplary systems: Amazon Comprehend, Google Cloud Services, Stanford Core NLP Pros: do not require great preprocessing of the data and allow the user to directly start a prediction "out of the box" Cons: limited fine-tuning for the underlying use-case (retraining might be needed to adjust the model performance)
Classical ML algorithms	**Deep Learning (DL) algorithms**
Application of traditional supervised classifiers like logistic regression, random forest, support vector machine or Naive Bayes Pros: well known, often financially and computationally cheap, easy to interpret Cons: in general, performance on unstructured data like text is expected to be worse compared to structured data and necessary preprocessing can be extensive	Application of NLP related neural network architectures like BERT, GPT-2 / GPT-3 mainly via transfer learning Pros: many pretrained neural networks for word embeddings and sentiment prediction already exist (particularly helpful for transfer learning), DL models scale effectively with data Cons: difficult and computationally expensive to tune architecture and hyperparameters

In this project, we focused on the latter two approaches since they are supposed to be the most promising. Thereby, we used SparkNLP as the NLP library of choice due to its extensive functionality, its scalability (fully supported by Apache Spark™) and accuracy (e.g., it contains multiple state-of-the-art embeddings and allows users to make use of transfer learning). First, we built a sentiment analysis pipeline using the aforementioned classical ML algorithms. The following figure shows its high-level architecture consisting of three parts: preprocessing, feature vectorization and finally training including hyperparameter tuning.

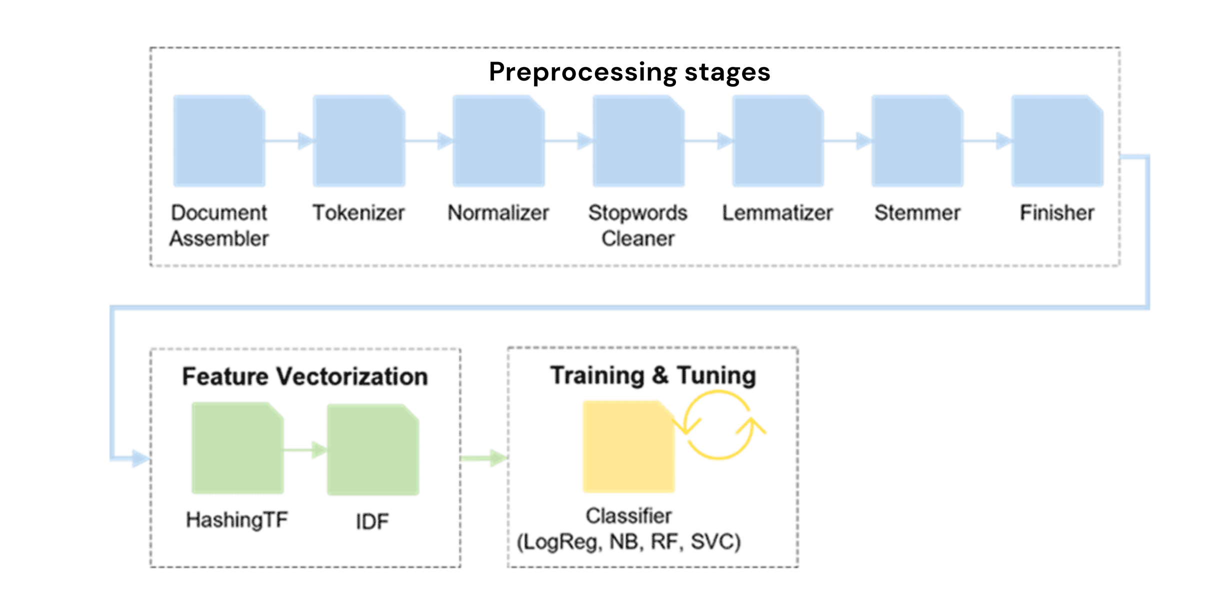


Figure 5: Machine learning model pipeline

We run this pipeline for every classifier and compare their corresponding accuracies on the test set. As a result, the Support Vector Classifier achieved the highest accuracy with 75.7%, closely followed by Logistic Regression (75.6%), Naïve Bayes (74%) and finally Random Forest (71.9%). To improve the performance, other supervised classifiers like XGBoost or GradientBoostedTrees could be tested. Besides, the individual algorithms could be combined to an ensemble, which is then used for prediction (e.g., majority voting, stacking).

In addition to this first pipeline, we developed a second Spark pipeline with a similar architecture making use of the rich Spark NLP functionalities regarding pretrained word embeddings and DL models. Starting with the standard Document Assembler annotator, we only used a Normalizer annotator to remove Twitter handles, alphanumeric characters, hyperlinks, html tags and timestamps but no further preprocessing related annotators. In terms of the training stage, we used a pretrained (on the well-known IMDb data set) sentiment DL model provided by Spark NLP. Using the default hyperparameter settings, we already achieved a test set accuracy of 83%, which could potentially be even enhanced using other pretrained word embeddings or sentiment DL models. Thus, the DL strategy clearly outperformed the pipeline in figure 5 with the Support Vector Classifier by around 7.4 percent points.

Correlation model

The project requirement included a correlation model on sentiment and price; therefore, we built a linear regression model using scikit-learn and mlflow.sklearn for this task.

We quantified the sentiment by assigning negative tweets a score of -1, neutral tweets a score of 0, and positive tweets a score of 1. The total sentiment score for each cryptocurrency is then calculated by adding up the scores for each cryptocurrency in 15-minute intervals. The linear regression model is built using the total sentiment score in each window for all companies to predict the percentage change in cryptocurrency prices. However, the model shows no clear linear relationship between sentiment and change in price. A possible future improvement for the correlation model is using sentiment polarity to predict the change in price instead.

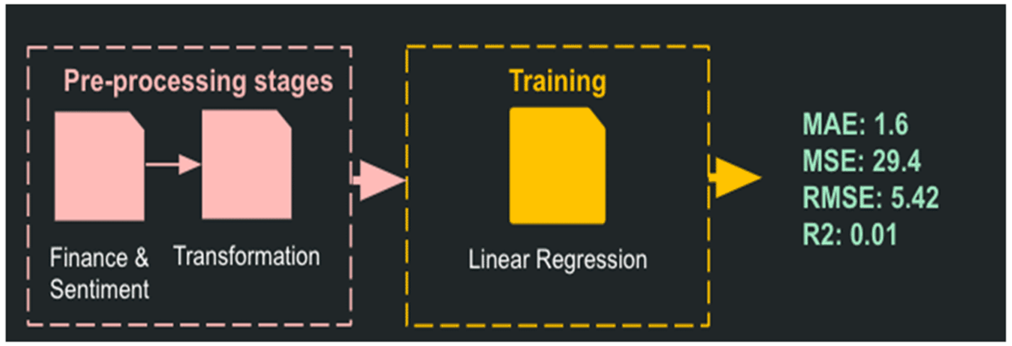


Figure 6: Correlation model pipeline

Business intelligence

Understanding stock correlation models was a key component of generating buy/sell predictions, but communicating results and interacting with the information is equally critical to make well-informed decisions. The market is so dynamic, so a real-time visualization is required to aggregate and organize trending information. Databricks Lakehouse enabled all of the BI analyst tasks to be coordinated in one place with streamlined access to the lakehouse data tables. First, a set of SQL queries were generated to extract and aggregate information from the lakehouse. Then the data tables were easily imported with a GUI tool to rapidly create dashboard views. In addition to the dashboards, alert triggers were created to notify users of critical activities like stock movement up/down by > X%, increases in Twitter activity about a particular crypto hashtag or changes in overall positive/negative sentiment about each cryptocurrency.

Dashboard generation

The business intelligence dashboards were created using Databricks SQL. This system provides a full ecosystem to generate SQL queries, create data views and charts, and ultimately organizes all of the information using Databricks Dashboards.

The use of the SQL Editor in Databricks was key to making the process fast and simple. For each query, the editor GUI enables the selection of different views of the data including tables, charts and summary statistics to immediately see the output. From there, views could be imported directly into the dashboards. This eliminated redundancy by utilizing the same query for different visualizations.

Visualization

For the topic of Twitter sentiment analysis, there are three key views to help users interact with the data on a deeper level.

View 1: Overview Page, taking a high-level view of Twitter influencers, stock movement and frequency of tweets related to particular cryptos.

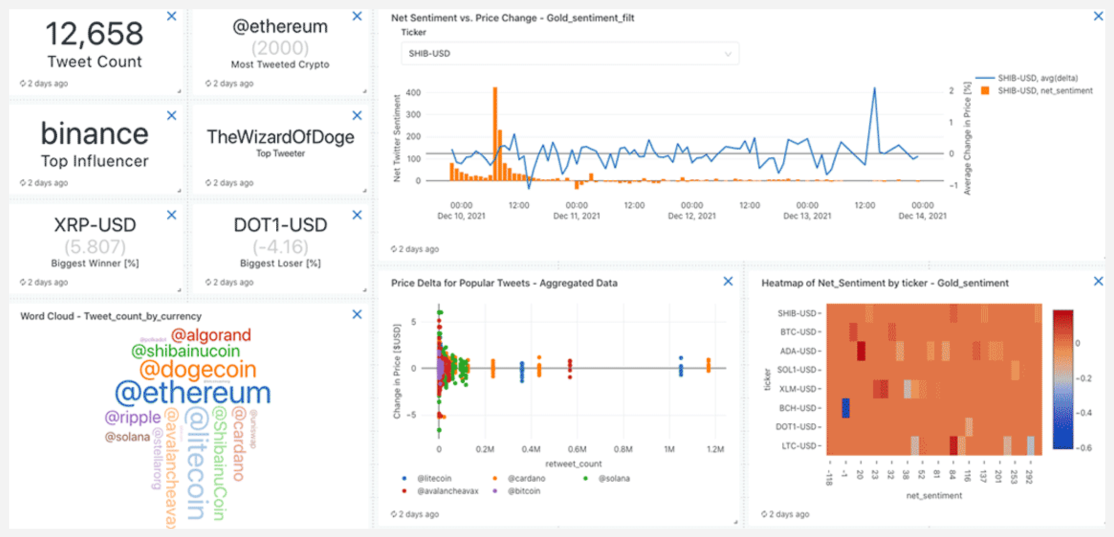


Figure 7: Overview Dashboard View with top level statistics

View 2: Sentiment Analysis, to understand whether each tweet is positive, negative or neutral. Here you can easily visualize which cryptocurrencies are receiving the most attention in a given time window.

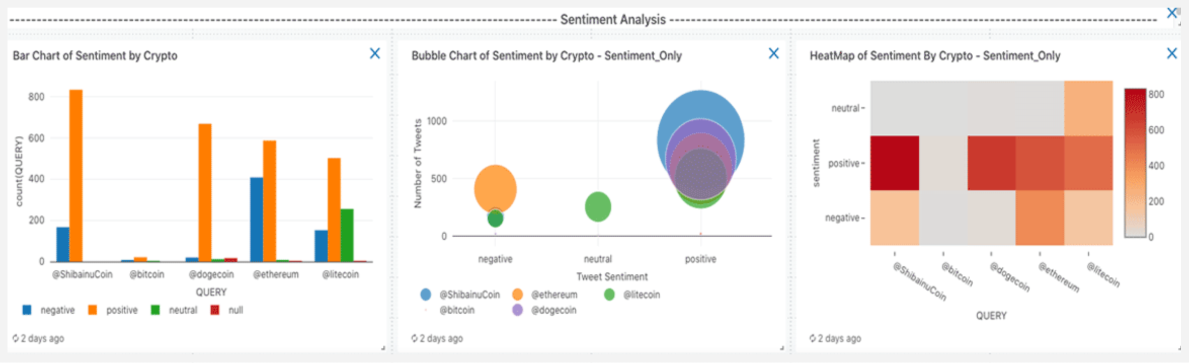


Figure 8: Sentiment Analysis Dashboard

View 3: Stock Volatility to provide the user with more specific information about the price for each cryptocurrency with trends over time.

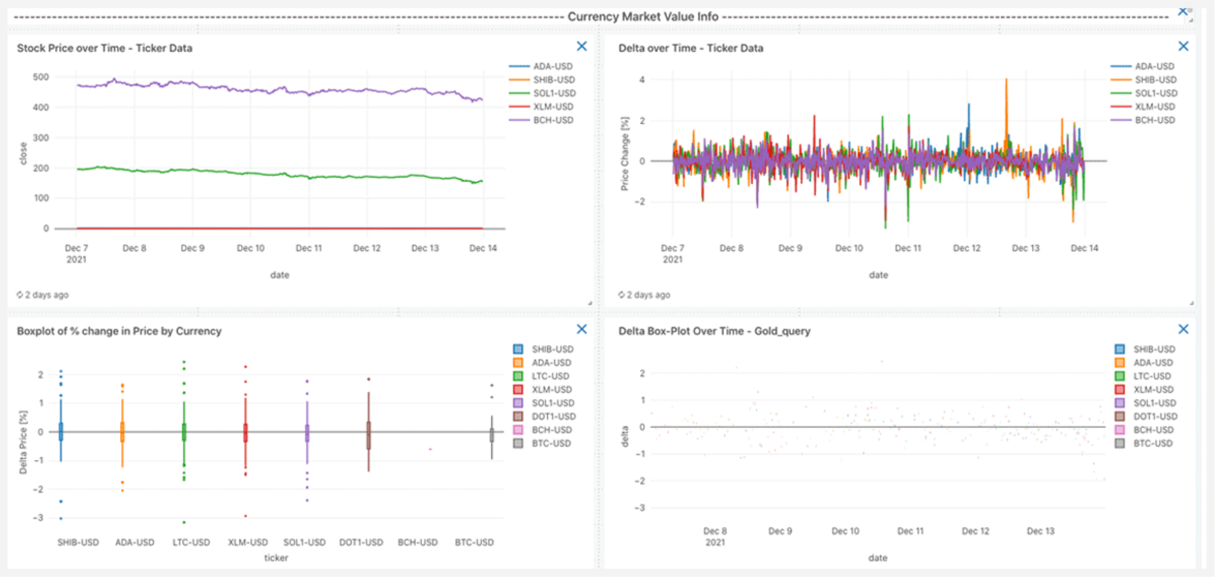


Figure 9: Stock Ticker Dashboard

Summary

Our team of data engineers, data scientists and BI analysts was able to leverage the Databricks tools to investigate the complex issue of Twitter usage and cryptocurrency stock movement. The lakehouse design created a robust data environment with smooth ingestion, processing and retrieval by the whole team. The data collection and cleaning pipelines deployed using Delta tables were easily managed even at high update frequencies. The data was analyzed by a natural language sentiment model and a stock correlation model using MLflow, which made the organization of various model versions simple. Powerful analytics dashboards were created to view and interpret the results using built-in SQL and Dashboard features. The functionality of Databricks' end-to-end product tools removed significant technical barriers, which enabled the entire project to be completed in less than 4 weeks with minimal challenges. This approach could easily be applied to other technologies where streamlined data pipelines, machine learning and BI analytics can be the catalyst for a deeper understanding of your data.

Our findings

These are additional conclusions from the data analysis to highlight the extent of Twitter users' influence on the price of cryptocurrencies.

Volume of tweets correlated with volatility in cryptocurrency price

There is a clear correlation in periods of high tweet frequency to the movement of a cryptocurrency. Note that this happens before and after a stock price change, indicating some tweet frenzies precede price change and are likely influencing value, and others are in response to big shifts in price.

Twitter users with more followers don't actually have more influence on crypto stock price

This is often discussed in media events, particularly with lesser-known currencies. Some extreme influencers like Elon Musk gained a reputation for being able to drive enormous market swings with a small number of targeted tweets. While it is true that a single tweet can impact cryptocurrency price, there is not an underlying correlation between number of followers to movement of the currency price. There is also a slightly negative correlation to number of retweets vs. price movement, indicating the Twitter activity by influencers might have broader reach as it moves into other mediums like new articles rather than reaching directly to investors.

The Databricks platform was incredibly useful for solving complex problems like merging Twitter and stock data.

Overall, the use of Databricks to coordinate the pipeline from data ingestions, the lakehouse data structure and the BI reporting dashboards was hugely beneficial to completing this project efficiently. In a short period of time, the team was able to build the data pipeline, complete machine learning models and produce high-quality visualizations to communicate results. The infrastructure provided by the Databricks platform removed many of the technical challenges and enabled the project to be successful.

While this tool will not enable you to outwit the cryptocurrency markets, we strongly believe it will predict periods of increased volatility, which can be advantageous for specific investing conditions.

Disclaimer: This article takes no responsibility for financial investment decisions. Nothing contained in this website should be construed as investment advice.

Try notebooks

Please try out the referenced Databricks notebooks

Data Science →

Orchestrator →

Tweepy →

Merge to Gold →

Inference →

Y_Finance →

CHAPTER 10: 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 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 10: 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 could 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

CHAPTER 10: 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 10: CUSTOMER CASE STUDIES

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

"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

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

CHAPTER 10: CUSTOMER CASE STUDIES

Showtime leverages ML to deliver data-driven content programming

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

“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

CHAPTER 10: CUSTOMER CASE STUDIES

Shell innovates with energy solutions for a cleaner world

"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 Advanced Analytics CoE
Shell

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

CHAPTER 10: CUSTOMER CASE STUDIES

Riot Games leverages AI to engage gamers and reduce churn

"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

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

CHAPTER 10: 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 auto-scaling 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 Eneco'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

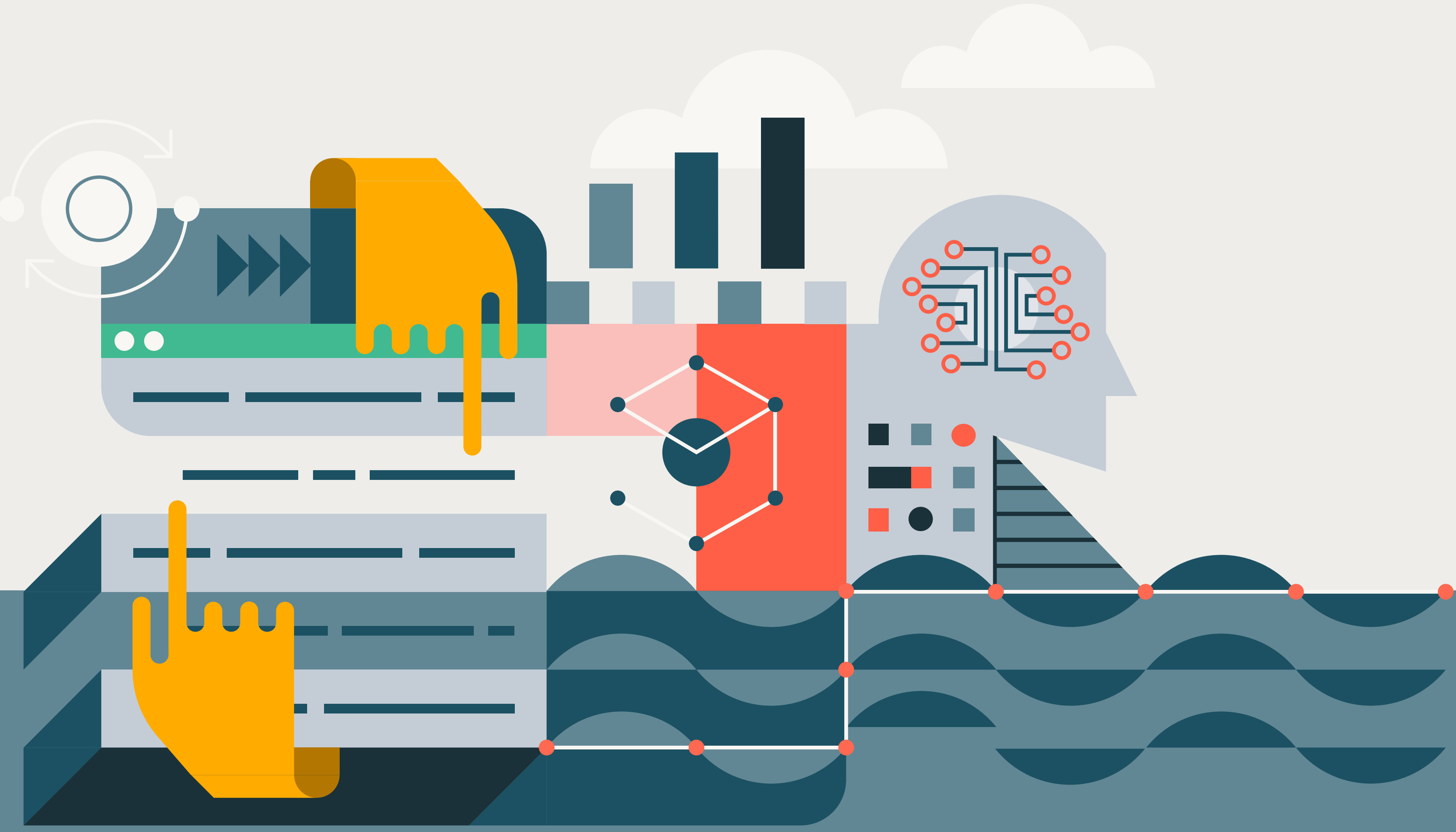
Learn more

About Databricks

Databricks is the lakehouse company. More than 7,000 organizations worldwide — including Comcast, Condé Nast, H&M and over 50% 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