eBook

The Big Book of Machine Learning Use Cases

A collection of technical blogs, including code samples and notebooks

databricks

Contents

CHAPTER 1:
Introduction 3

CHAPTER 2:
Using Dynamic Time Warping and MLflow to Detect Sales Trends

Part 1: Understanding Dynamic Time Warping 4

Part 2: Using Dynamic Time Warping and MLflow to Detect Sales Trends 10

CHAPTER 3:
Fine-Grain Time Series Forecasting at Scale With Prophet and Apache Spark 18

CHAPTER 4:
Doing Multivariate Time Series Forecasting With Recurrent Neural Networks 25

CHAPTER 5:
Detecting Financial Fraud at Scale With Decision Trees and MLflow on Databricks 31

CHAPTER 6:
Automating Digital Pathology Image Analysis With Machine Learning on Databricks 42

CHAPTER 7:
A Convolutional Neural Network Implementation for Car Classification 48

CHAPTER 8:
Processing Geospatial Data at Scale With Databricks 56

CHAPTER 9:
Customer Case Studies 70

CHAPTER 1:

Introduction

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

CHAPTER 2:

Understanding Dynamic Time Warping

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

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

April 30, 2019

Try this notebook in Databricks →

Introduction

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

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

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

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

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

In this blog series, we will explore:

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

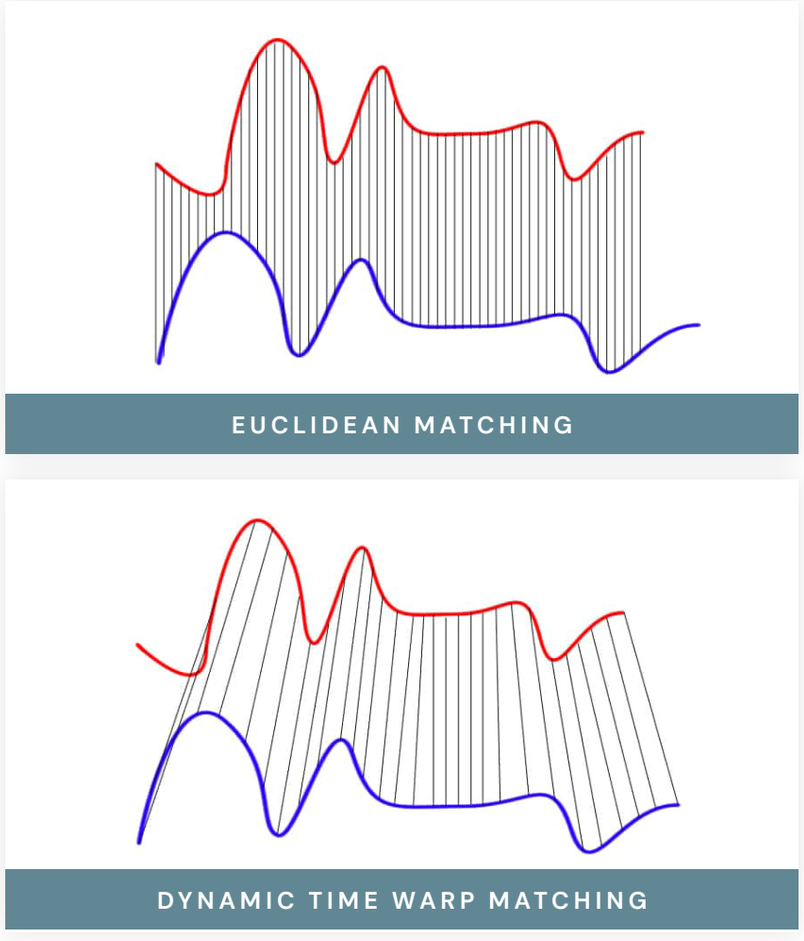
Dynamic time warping

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

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

Background

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



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]

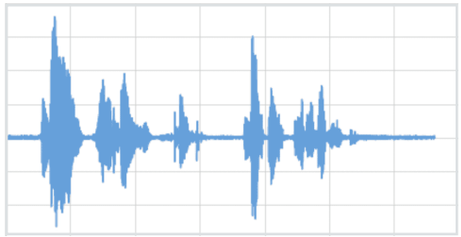
0:00 / 0:07

Quotes are from "The Expanse"

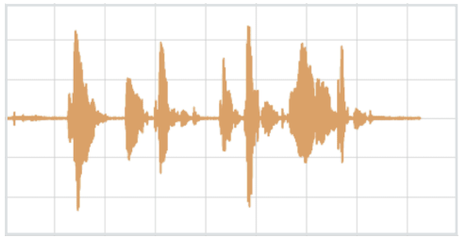
Below are visualizations using `matplotlib` of the four audio clips:

- **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.

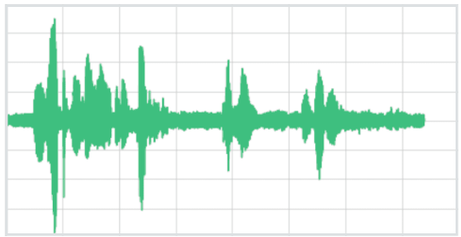
Clip 1 Doors and corners, kid. That's where they get you. [v1]



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.



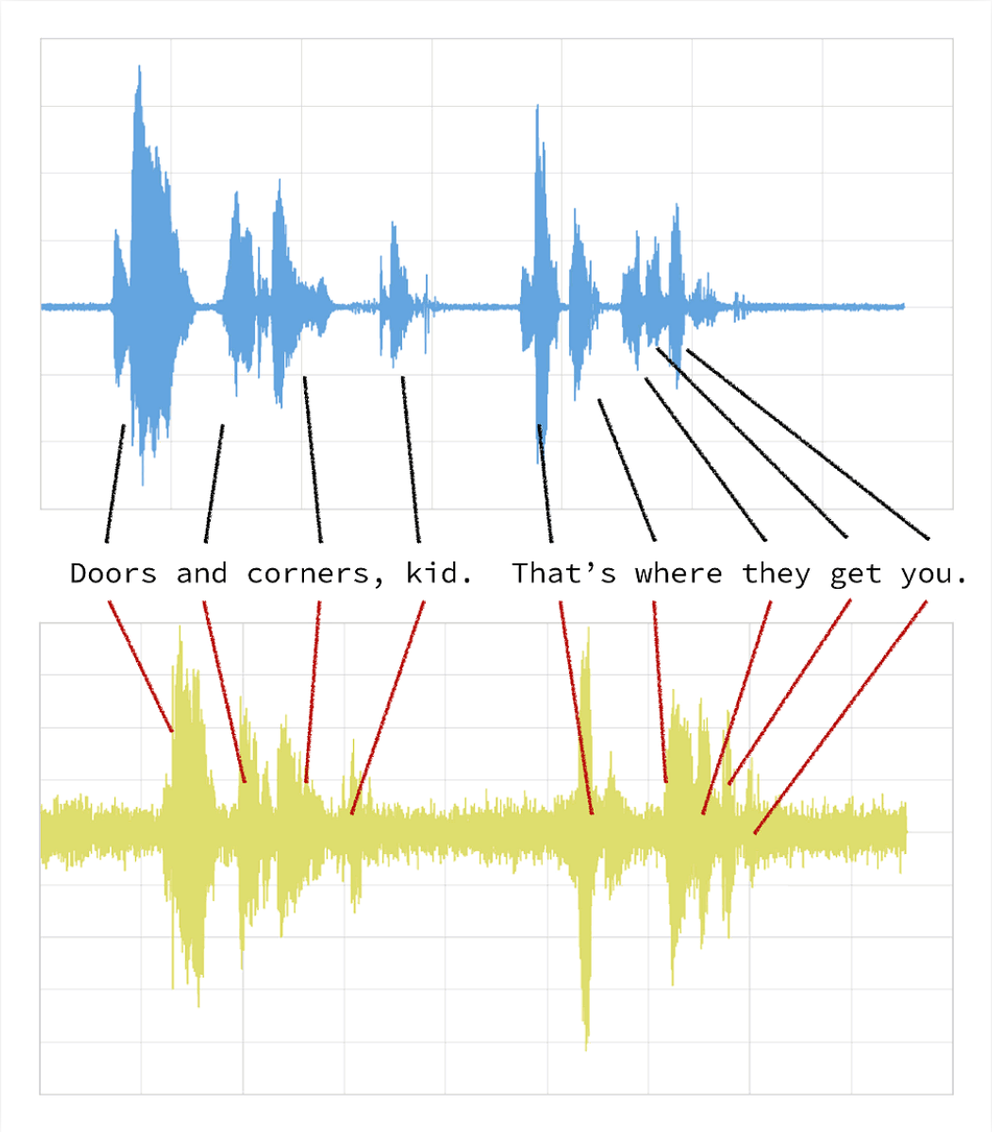
Clip 4 Doors and corners, kid. That's where they get you. [v3]

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

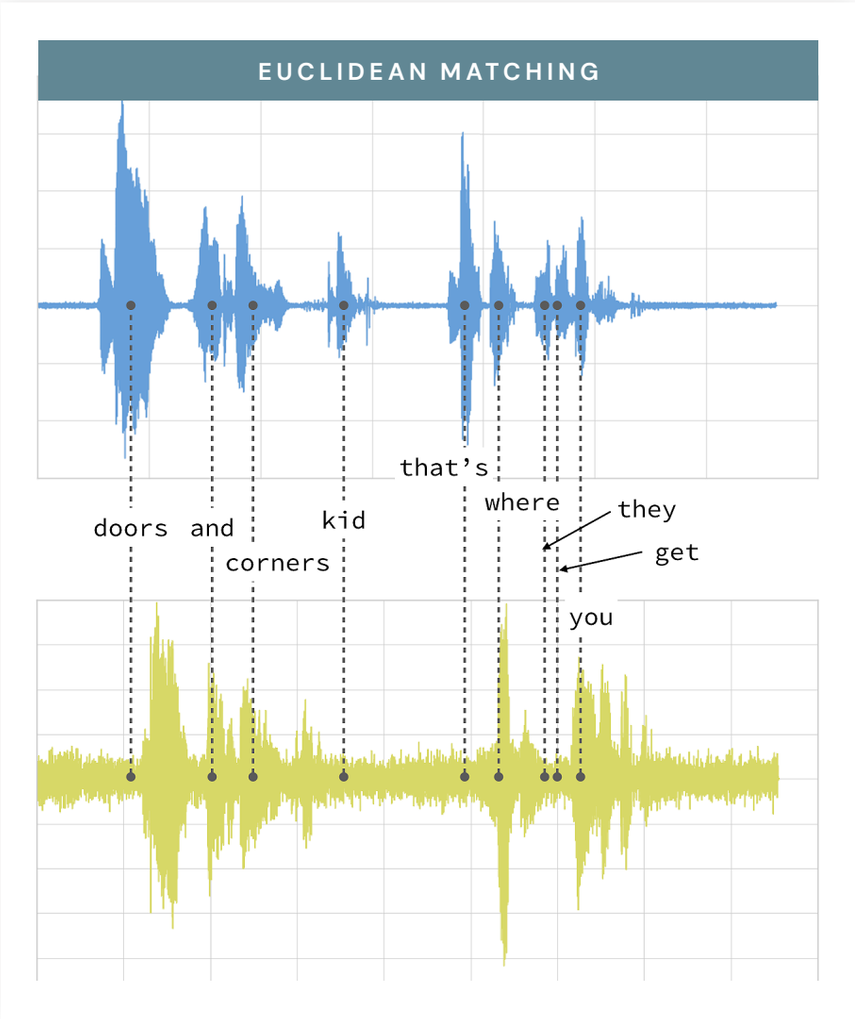
```
from scipy.io import wavfile
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure

# 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.

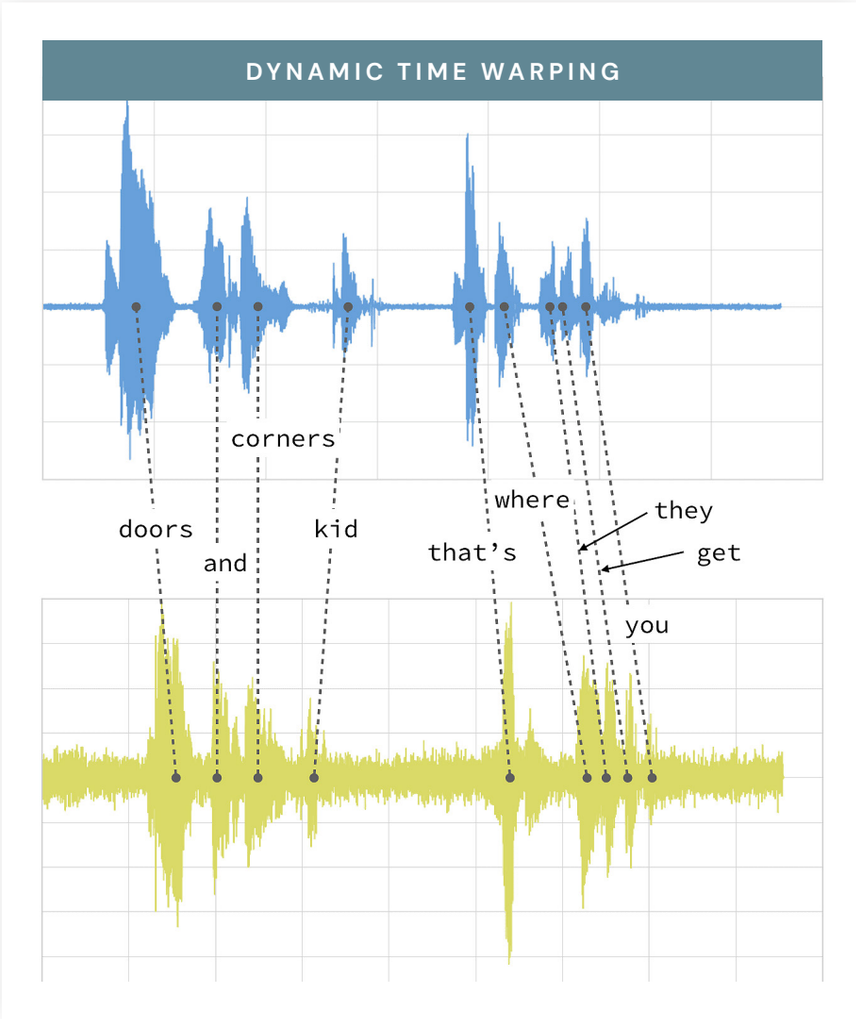
As noted below, the two clips (in this case, clips 1 and 4) have different intonations (amplitude) and latencies for the same quote.



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.

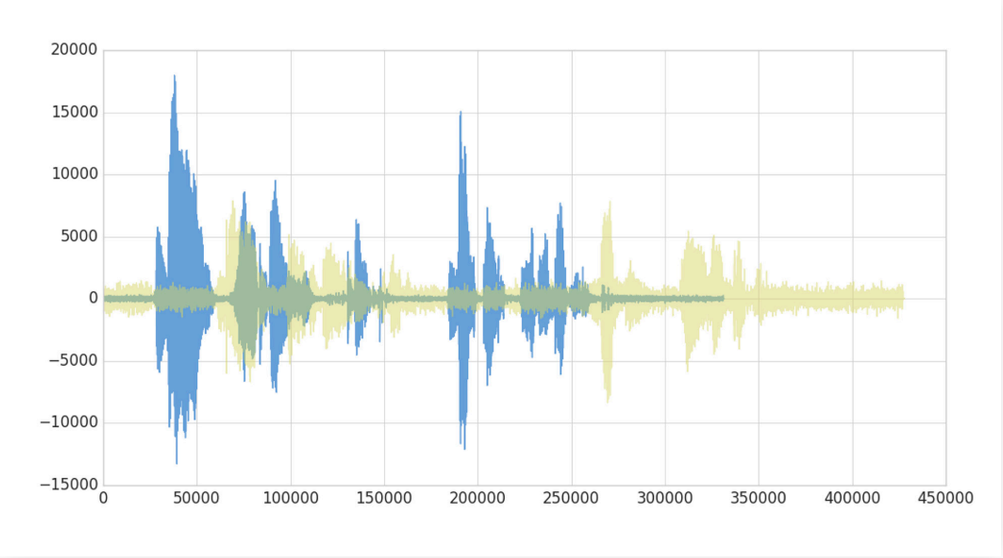


For our time series comparison, we will use the `fastdtw` PyPi library; the instructions to install PyPi libraries within your Databricks workspace can be found here: Azure | AWS. By using fastdtw, we can quickly calculate the distance between the different time series.

```
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 2:

Using Dynamic Time Warping and MLflow to Detect Sales Trends

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

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

April 30, 2019

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

Background

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

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

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

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

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

Load the product sales data set

We will use the weekly sales transaction data set found in the UCI Data Set Repository to perform our sales-based time series analysis. (Source attribution: James Tan, jamestansc@suss.edu.sg, Singapore University of Social Sciences)

```
import pandas as pd

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

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

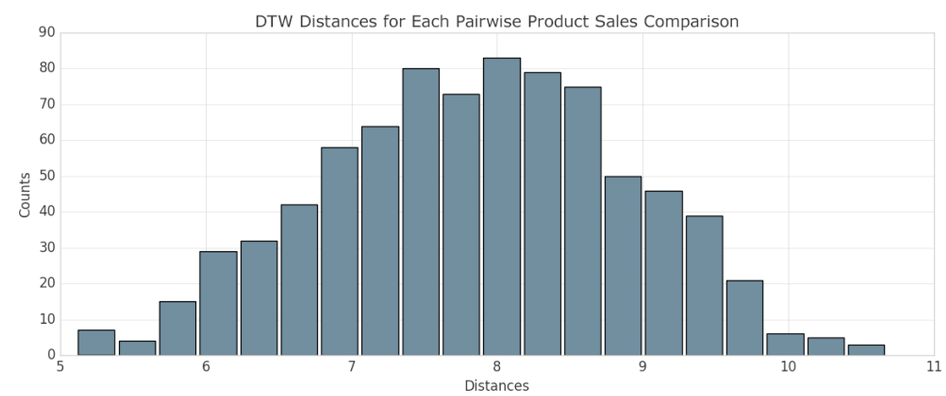
Product_Code	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
P1	11	12	10	8	13	12	14	21	6	14	11	14	16	9
P2	7	6	3	2	7	1	6	3	3	3	2	2	6	2
P3	7	11	8	9	10	8	7	13	12	6	14	9	4	7
P4	12	8	13	5	9	6	9	13	13	11	8	4	5	4
P5	8	5	13	11	6	7	9	14	9	9	11	18	8	4
P6	3	3	2	7	6	3	8	6	6	3	1	1	5	4
P7	4	8	3	7	8	7	2	3	10	3	5	2	3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

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

Calculate distance to optimal time series by product code

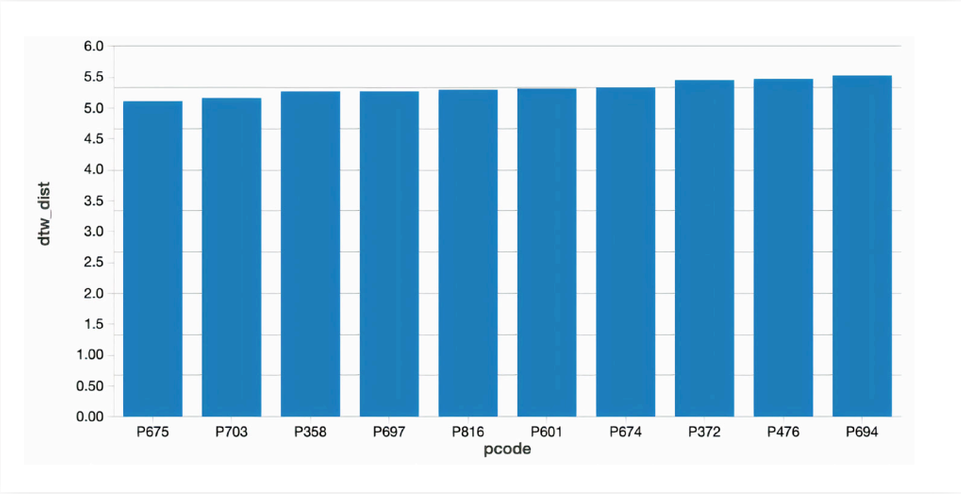
```
# Calculate distance via dynamic time warping between product
code and optimal time series
import numpy as np
import _ucrdtw

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

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

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

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

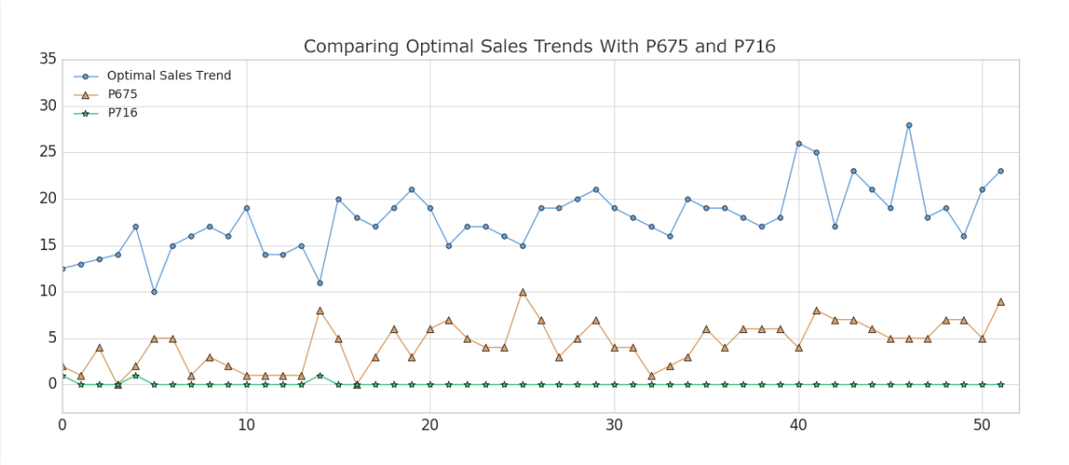


From there, we can identify the product codes closest to the optimal sales trend (i.e., those that have the smallest calculated DTW distance). Since we're using Databricks, we can easily make this selection using a SQL query. Let's display those that are closest.

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



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

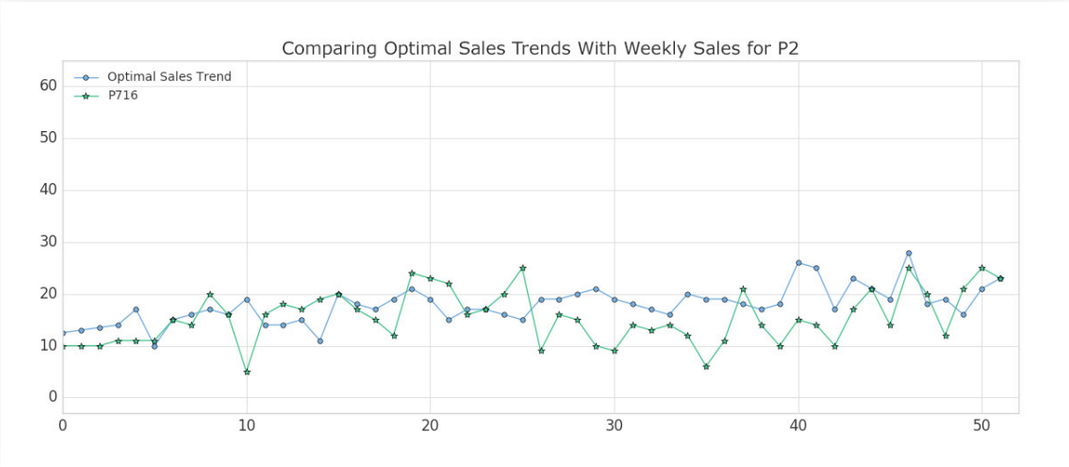


As you can see, Product #675 (shown in the orange triangles) represents the best match to the optimal sales trend, although the absolute weekly sales are lower than we'd like (we'll remedy that later). This result makes sense since we'd expect the product with the closest DTW distance to have peaks and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

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

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

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



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

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

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

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

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

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

as shown in the abbreviated code at right.

```
import mlflow

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

        # Log Model using Custom Flavor
        dtw_model = {'stretch_factor' : float(ts_stretch_factor),
'pattern' : optimal_pattern}
        mlflow_custom_flavor.log_model(dtw_model, artifact_path="model")

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

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

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

    return run.info

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

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

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

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

Logging models in MLflow

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

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

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

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

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

Loading a logged model from one of our runs

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

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

```
import custom_flavor as mlflow_custom_flavor

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

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

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

Next steps

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

CHAPTER 3:

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

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

January 27, 2020

Try this time series forecasting notebook in Databricks →

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

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

Accurate and timely forecasting is now more important than ever

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

New expectations require more precise time series forecasting methods and models

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

PROPHET

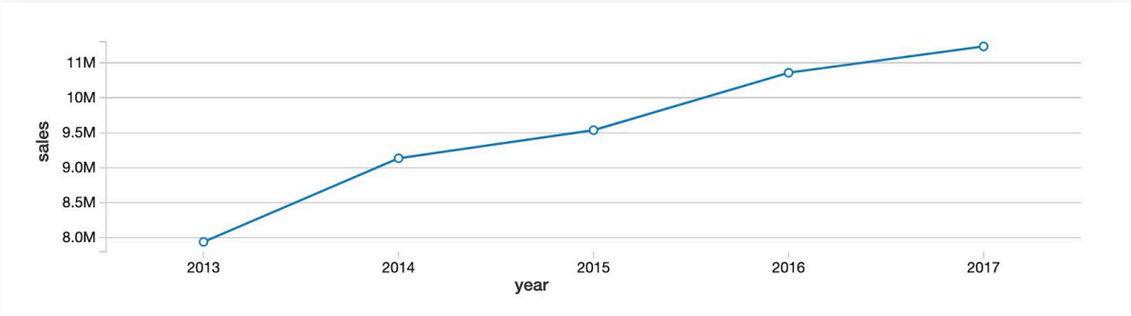
New capabilities, such as those provided by Facebook Prophet, are emerging from the data science community, and companies are seeking the flexibility to apply these machine learning models to their time series forecasting needs.

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

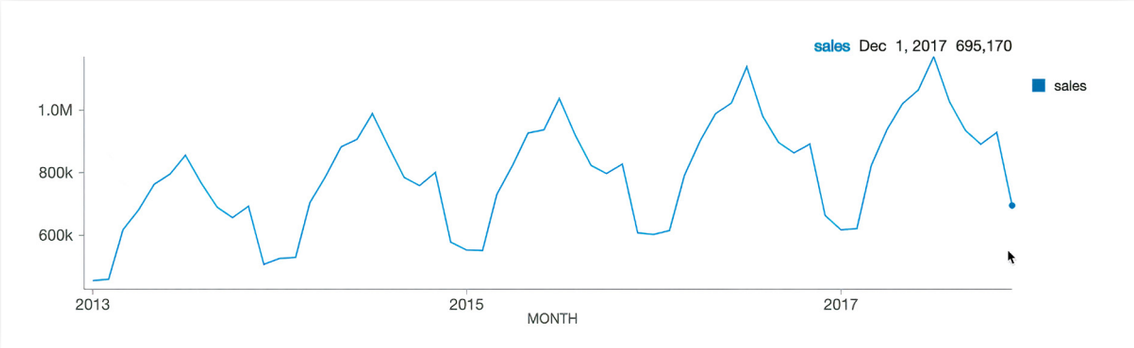
Visualizing demand seasonality in time series data

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

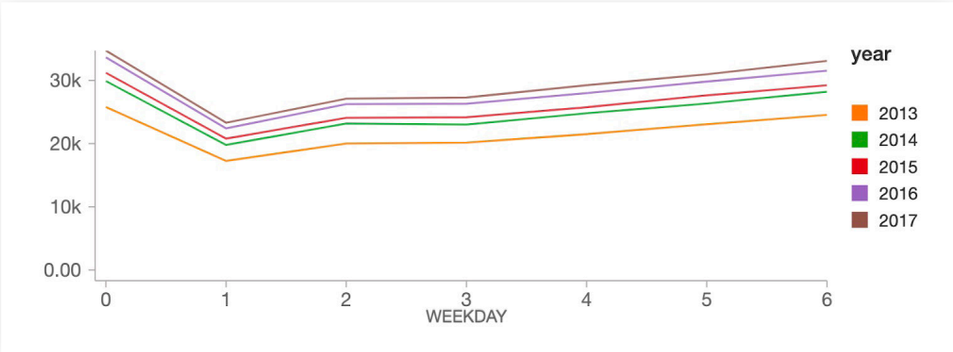
To get started, let's look at the overall yearly sales trend for all products and stores. As you can see, total product sales are increasing year over year with no clear sign of convergence around a plateau.



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



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



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

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

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

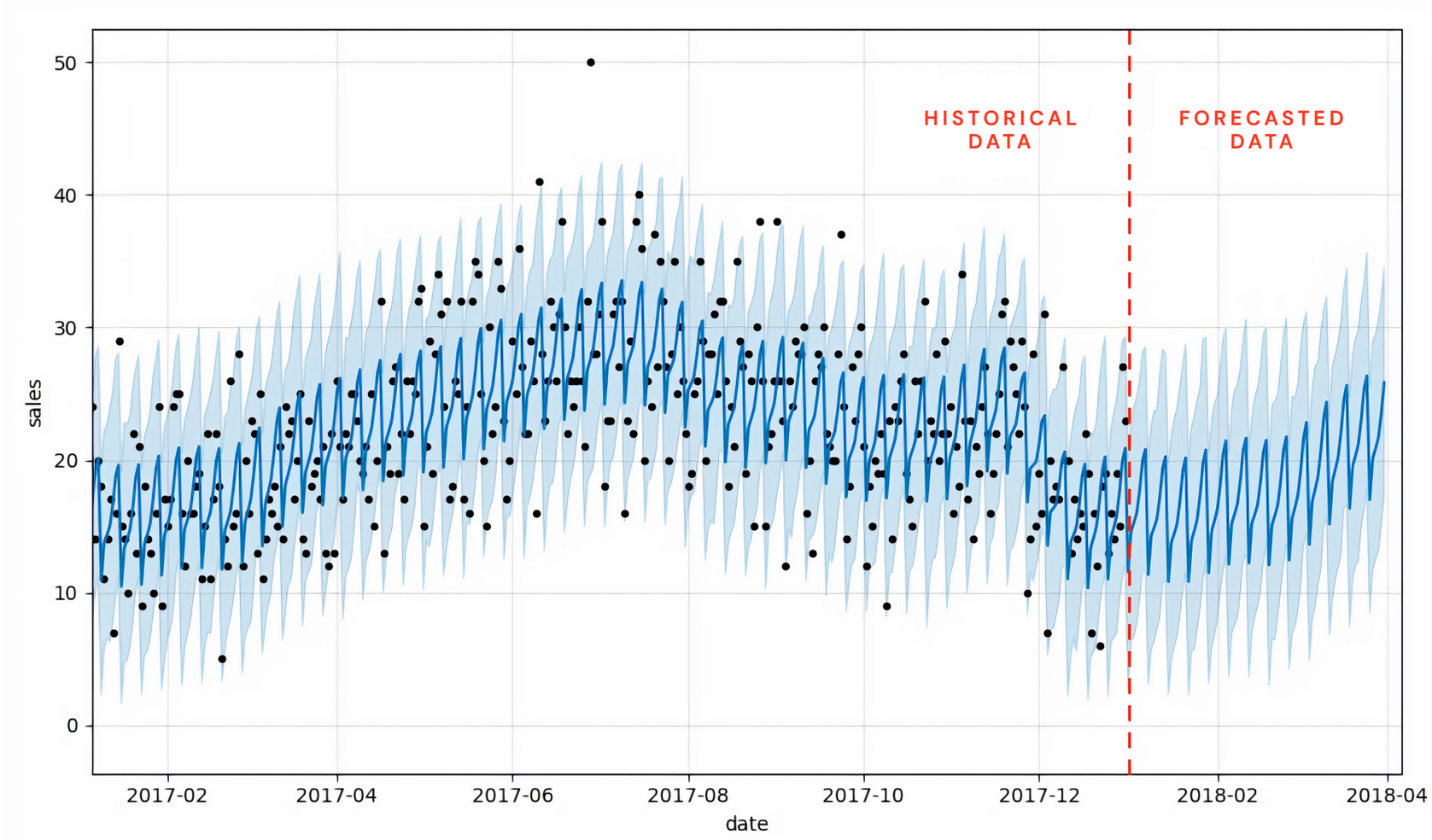
```
import pandas as pd
from fbprophet import Prophet

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

Leveraging the power of pandas user-defined functions

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

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

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

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

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

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

    # fit the model
    model.fit(history_pd)

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

    # make predictions
    results_pd = model.predict(future_pd)

    # . . .

    # return predictions
    return results_pd
```

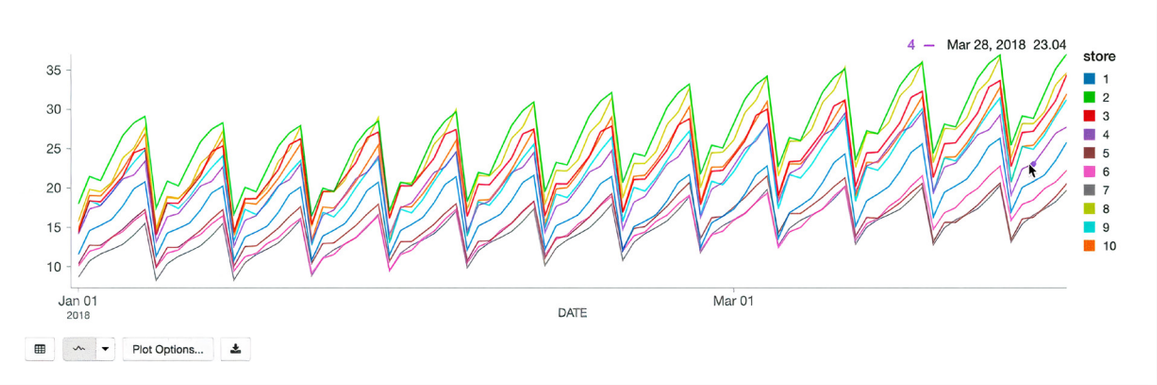
Now, to bring it all together, we use the `groupBy` command we discussed earlier to ensure our data set is properly partitioned into groups representing specific store and item combinations. We then simply `apply` the UDF to our DataFrame, allowing the UDF to fit a model and make predictions on each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

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

```
from pyspark.sql.functions import current_date

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

Next steps

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



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

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

CHAPTER 4:

Doing Multivariate Time Series Forecasting With Recurrent Neural Networks

Using Keras' implementation of Long Short-Term Memory (LSTM) for time series forecasting

By **Vedant Jain**

September 10, 2019

Try this notebook in Databricks →

Time series forecasting is an important area in machine learning. It can be difficult to build accurate models because of the nature of the time-series data. With recent developments in the neural networks aspect of machine learning, we can tackle a wide variety of problems that were either out of scope or difficult to do with classical time series predictive approaches. In this post, we will demonstrate how to use Keras' implementation of Long Short-Term Memory (LSTM) for time series forecasting and MLfLow for tracking model runs.

What are LSTMs?

LSTM is a type of Recurrent Neural Network (RNN) that allows the network to retain long-term dependencies at a given time from many timesteps before. RNNs were designed to that effect using a simple feedback approach for neurons where the output sequence of data serves as one of the inputs. However, long-term dependencies can make the network untrainable due to the vanishing gradient problem. LSTM is designed precisely to solve that problem.

Sometimes accurate time series predictions depend on a combination of both bits of old and recent data. We have to efficiently learn even what to pay attention to, accepting that there may be a long history of data to learn from. LSTMs combine simple DNN architectures with clever mechanisms to learn what parts of history to "remember" and what to "forget" over long periods. The ability of LSTMs to learn patterns in data over long sequences makes them suitable for time series forecasting.

For the theoretical foundation of LSTM's architecture, see here (Chapter 4).

Choose the right problem and right data set

There are innumerable applications of time series — from creating portfolios based on future fund prices to demand prediction for an electricity supply grid and so on. In order to showcase the value of LSTM, we first need to have the right problem and more importantly, the right data set. Say we want to learn to predict humidity and temperature in a house ahead of time so a smart sensor can proactively turn on the A/C, or you just want to know the amount of electricity you will consume in the future so you can proactively cut costs. Historical sensor and temperature data ought to be enough to learn the relationship, and LSTMs can help, because it won't just depend on recent sensor values, but more importantly older values, perhaps sensor values from the same time on the previous day. For this purpose, we will use experimental data about appliances' energy use in a low-energy building.

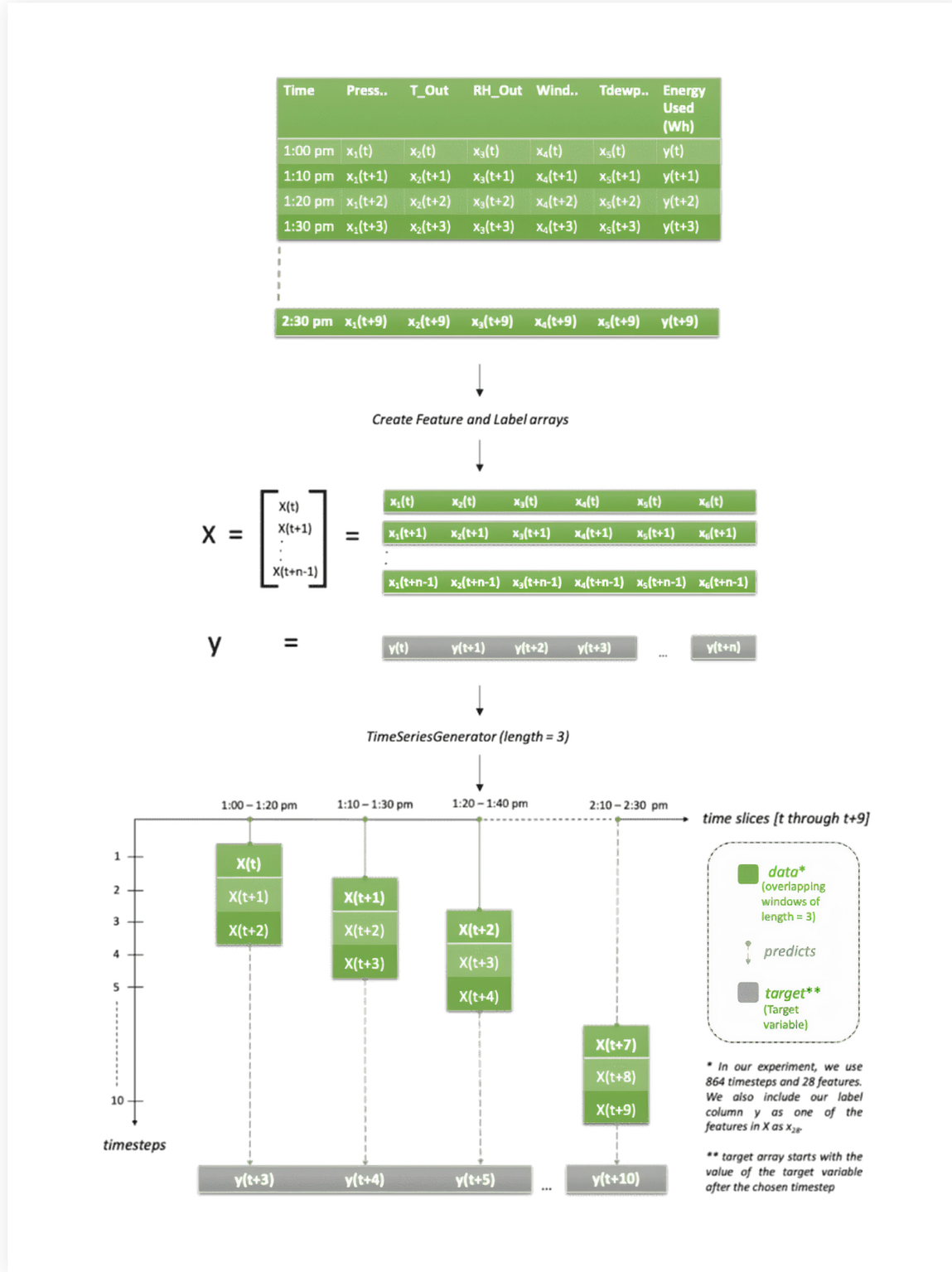
Experiment

The data set we chose for this experiment is perfect for building regression models of appliances' energy use. The house temperature and humidity conditions were monitored with a ZigBee wireless sensor network. It is at 10-minute intervals for about 4.5 months. The energy data was logged with m-bus energy meters. Weather from the nearest airport weather station (Chievres Airport, Belgium) was downloaded from a public data set from Reliable Prognosis (rp5.ru), and merged together with the experimental data sets using the date and time column. The data set can be downloaded from the UCI Machine Learning repository.

We'll use this to train a model that predicts the energy consumed by household appliances for the next day.

Data modeling

Before we can train a neural network, we need to model the data in a way the network can learn from a sequence of past values. Specifically, LSTM expects the input data in a specific 3D tensor format of test sample size by timesteps by the number of input features. As a supervised learning approach, LSTM requires both features and labels in order to learn. In the context of time series forecasting, it is important to provide the past values as features and future values as labels, so LSTMs can learn how to predict the future. Thus, we explode the time series data into a 2D array of features called 'X', where the input data consists of overlapping lagged values at the desired number of timesteps in batches. We generate a 1D array called 'y' consisting of only the labels or future values that we are trying to predict for every batch of input features. The input data also should include lagged values of 'y' so the network can also learn from past values of the labels. See the following image for further explanation:



Our data set has 10-minute samples. In the image above, we have chosen length = 3, which implies we have 30 minutes of data in every sequence (at 10-minute intervals). By that logic, features 'X' should be a tensor of values [X(t), X(t+1), X(t+2)], [X(t+2), X(t+3), X(t+4)], [X(t+3), X(t+4), X(t+5)]...and so on. And our target variable 'y' should be [y(t+3), y(t+4), y(t+5)...y(t+10)] because the number of timesteps or length is equal to 3, so we will ignore values y(t), y(t+1), y(t+2). Also, in the graph, it's apparent that for every input row, we're only predicting one value out in the future, i.e., y(t+n+1); however, for more realistic scenarios you can choose to predict further out in the future, i.e., y(t+n+L), as you will see in our example below.

The Keras API has a built-in class called TimeSeriesGenerator that generates batches of overlapping temporal data. This class takes in a sequence of data points gathered at equal intervals, along with time series parameters such as stride, length of history, etc. to produce batches for training/validation.

So, let's say for our use case, we want to learn to predict from 6 days' worth of past data and predict values some time out in the future, let's say, 1 day. In that case, length is equal to 864, which is the number of 10-minute timesteps in 6 days (24x6x6). Similarly, we also want to learn from past values of humidity, temperature, pressure, etc., which means that for every label we will have 864 values per feature. Our data set has a total of 28 features. When generating the temporal sequences, the generator is configured to return batches consisting of 6 days' worth of data every time. To make it a more realistic scenario, we choose to predict the usage 1 day out in the future (as opposed to the next 10-minute time interval), we prepare the test and train data set in a manner that the target vector is a set of values 144 timesteps (24x6x1) out in the future. For details, see the notebook, section 2: Normalize and prepare the data set.

The shape of the input set should be (samples, timesteps, input_dim) [https://keras.io/layers/recurrent/]. For every batch, we will have all 6 days' worth of data, which is 864 rows. The batch size determines the number of samples before a gradient update takes place.

```
# Create overlapping windows of lagged values for training and testing
datasets
timesteps = 864
train_generator = TimeseriesGenerator(trainX, trainY, length=timesteps,
sampling_rate=1, batch_size=timesteps)
test_generator = TimeseriesGenerator(testX, testY, length=timesteps,
sampling_rate=1, batch_size=timesteps)
```

For a full list of tuning parameters, see here.

Model training

LSTMs are able to tackle the long-term dependency problems in neural networks, using a concept known as Backpropogation through time (BPTT). To read more about BPTT, see here.

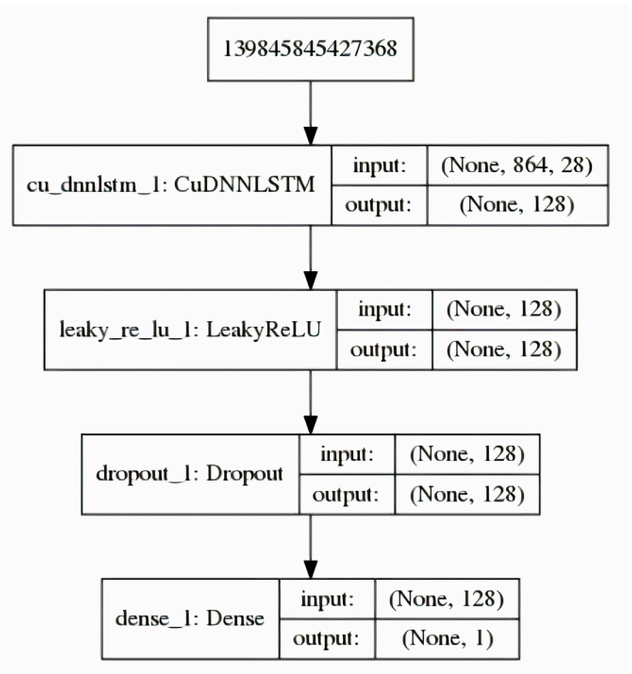
Before we train a LSTM network, we need to understand a few key parameters provided in Keras that will determine the quality of the network.

1. **Epochs:** Number of times the data will be passed to the neural network
2. **Steps per epoch:** The number of batch iterations before a training epoch is considered finished
3. **Activations:** Layer describing which activation function to use
4. **Optimizer:** Keras provides built-in optimizers

```
units = 128
num_epoch = 5000
learning_rate = 0.00144

with mlflow.start_run(experiment_id=3133492, nested=True):

  model = Sequential()
  model.add(CuDNNLSTM(units, input_shape=(train_X.shape[1], train_X.
  shape[2])))
  model.add(LeakyReLU(alpha=0.5))
  model.add(Dropout(0.1))
  model.add(Dense(1))

adam = Adam(lr=learning_rate)
# Stop training when a monitored quantity has stopped improving.
callback = [EarlyStopping(monitor="loss", min_delta = 0.00001, patience =
50, mode = 'auto', restore_best_weights=True), tensorboard]

# Using regression loss function 'Mean Standard Error' and validation
metric 'Mean Absolute Error'
model.compile(loss="mse", optimizer=adam, metrics=['mae'])
     )
```

In order to take advantage of the speed and performance of GPUs, we use the CUDNN implementation of LSTM. We have also chosen an arbitrarily high number of epochs. This is because we want to make sure that the data undergoes as many iterations as possible to find the best model fit. As for the number of units, we have 28 features, so we start with 32. After a few iterations, we found that using 128 gave us decent results.

For choosing the number of epochs, it's a good approach to choose a high number to avoid underfitting. In order to circumvent the problem of overfitting, you can use built in callbacks in Keras API; specifically EarlyStopping. EarlyStopping stops the model training when the monitored quantity has stopped improving. In our case, we use loss as the monitored quantity and the model will stop training when there's no decrease of 1e-5 for 50 epochs. Keras has built-in regularizers (weighted, dropout) that penalize the network to ensure a smoother distribution of parameters, so the network does not rely too much on the context passed between the units. See image below for layers in the network.



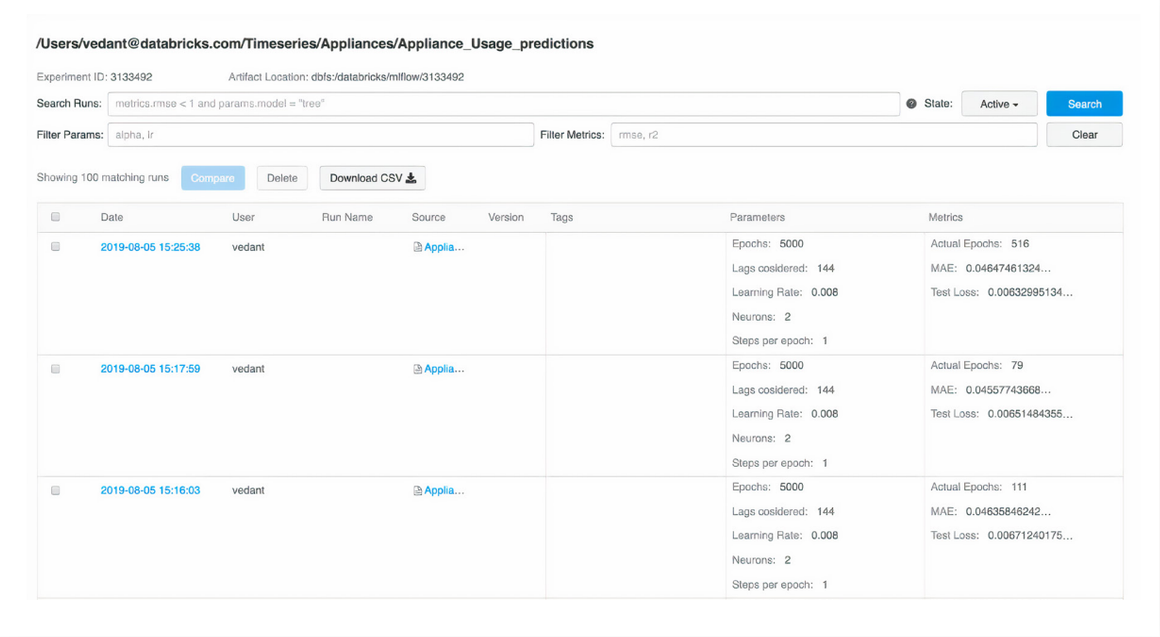
In order to send the output of one layer to the other, we need an activation function. In this case, we use LeakyRelu, which is a better variant of its predecessor, the Rectifier Linear Unit or Relu for short.

Keras provides many different optimizers for reducing loss and updates weights iteratively over epochs. For a full list of optimizers, see here. We choose the Adam version of stochastic gradient descent.

An important parameter of the optimizer is learning_rate, which can determine the quality of the model in a big way. You can read more about the learning rate here. We experimented with various values such as 0.001(default), 0.01, 0.1, etc. and found that 0.00144 gave us the best model performance in terms of speed of training and minimal loss. You can also use the LearningRateSchedular callback in order to tweak the learning rate to the optimal value. We used MLflow to track and compare results across multiple model runs.

Model evaluation and logging using MLflow

As you can see Keras implementation of LSTMs takes in quite a few hyperparameters. In order to find the best model fit, you will need to experiment with various hyperparameters, namely units, epochs, etc. You will also want to compare past model runs and measure model behavior over time and changes in data. MLflow is a great tool with an easy-to-use UI that allows you to do the above and more. Here you can see how easy it is to use MLflow to develop with Keras and TensorFlow, log an MLflow run and track experiments over time.



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. They can compare two or more model runs to understand the impact of various hyperparameters, until they conclude on the most optimal model.

The data engineers will then be able to easily retrieve the chosen model along with the library versions used for training to be deployed on new data in production. The final model can be persisted with the python_function flavor. It can then be used as an Apache Spark UDF, which once uploaded to a Spark cluster, will be used to score future data. You can find the full list of model flavors supported by MLflow here.

Summary

- LSTM can be used to learn from past values in order to predict future occurrences. LSTMs for time series don't make certain assumptions that are made in classical approaches, so it makes it easier to model time series problems and learn nonlinear dependencies among multiple inputs.
- When creating a sequence of events before feeding into the LSTM network, it is important to lag the labels from inputs, so the LSTM network can learn from past data. TimeSeriesGenerator class in Keras allows users to prepare and transform the time series data set with various parameters before feeding the time lagged data set to the neural network.
- LSTM has a series of tunable hyperparameters, such as epochs, batch size, etc., which are imperative to determining the quality of the predictions. Learning rate is an important hyperparameter that controls how the model weights get updated and the speed at which the model learns. It is very important to determine an optimal value for the learning rate in order to get the best model performance. Consider using the LearingRateSchedular callback parameter in order to tweak the learning rate to the optimal value.
- Keras provides a choice of different optimizers to use with respect to the type of problem you're solving. Generally, Adam tends to do well. Using MLflow UI, the user can compare model runs side by side to choose the best model.
- For time series, it's important to maintain temporality in the data so the LSTM network can learn patterns from the correct sequence of events. Therefore, it is important not to shuffle the data when creating test and validation sets and also when fitting the model.
- Like all machine learning approaches, LSTM is not immune to bad fitting, which is why Keras has EarlyStopping callback. With some degree of intuition and the right callback parameters, you can get decent model performance without putting too much effort in tuning hyperparameters.
- RNNs, specifically LSTMs, work best when given large amounts of data. So, when little data is available, it is preferable to start with a smaller network with a few hidden layers. Smaller data also allows users to provide a larger batch of data to every epoch, which can yield better results.

CHAPTER 5:

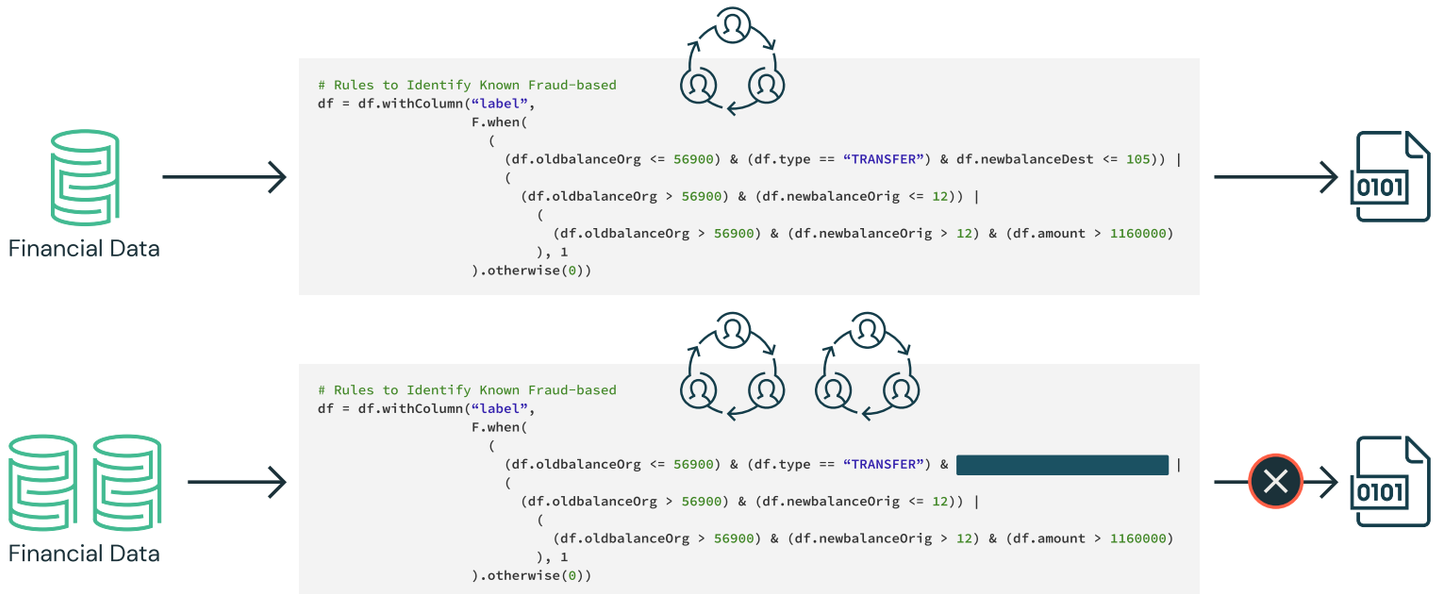
Detecting Financial Fraud at Scale With Decision Trees and MLflow on Databricks

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

May 2, 2019

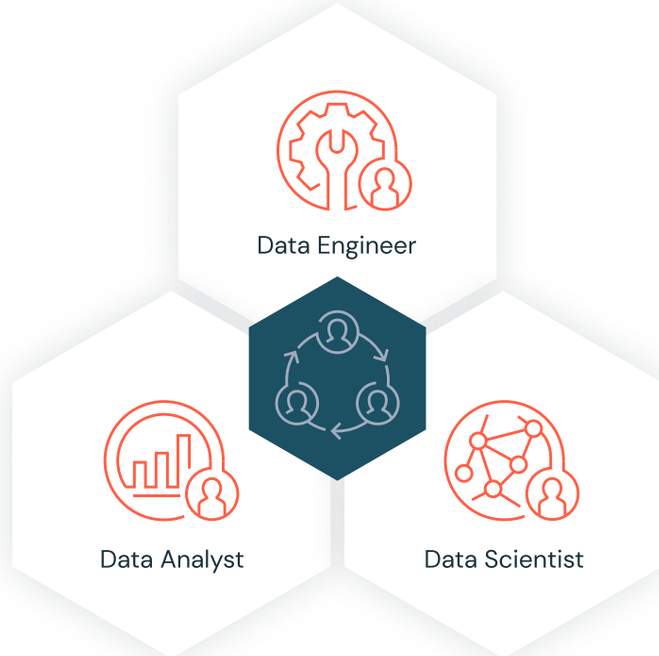
Try this notebook in Databricks →

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



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

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

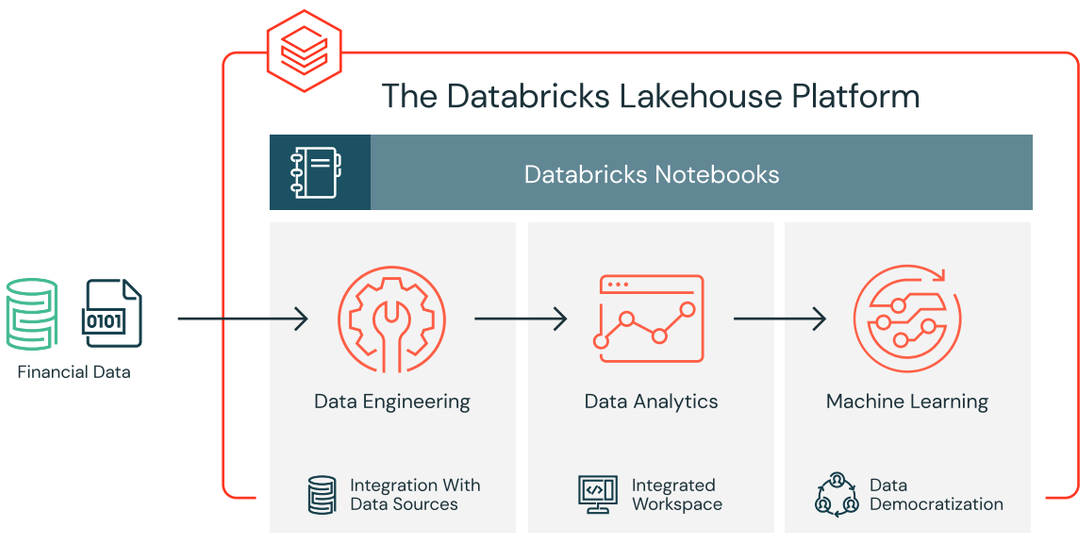


Additionally, the systems in the workflow described above are often siloed, with the domain experts, data scientists and data engineers all compartmentalized. The data engineer is responsible for maintaining massive amounts of data and translating the work of the domain experts and data scientists into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

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

Solving with machine learning

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



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

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

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

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

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

Ingesting and exploring the data

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

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

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

Exploring the data

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

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

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

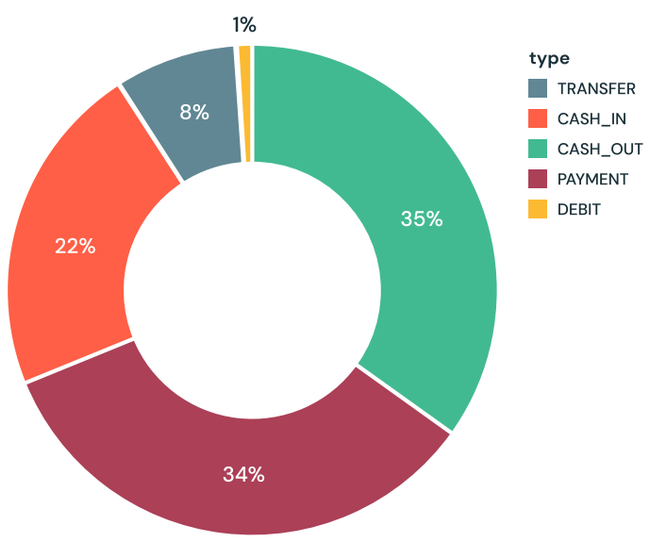
```
# Review the schema of your data
df.printSchema()
root
 |-- step: integer (nullable = true)
 |-- type: string (nullable = true)
 |-- amount: double (nullable = true)
 |-- nameOrig: string (nullable = true)
 |-- oldbalanceOrg: double (nullable = true)
 |-- newbalanceOrig: double (nullable = true)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

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

Types of transactions

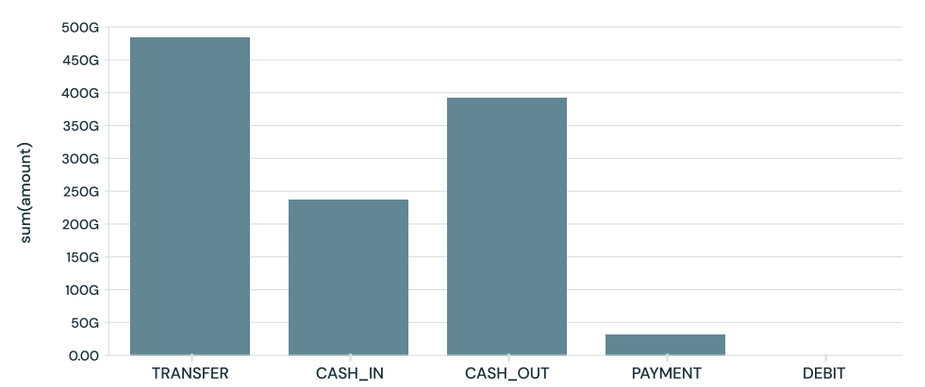
Let's visualize the data to understand the types of transactions the data captures and their contribution to the overall transaction volume.

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



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

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



Rules-based model

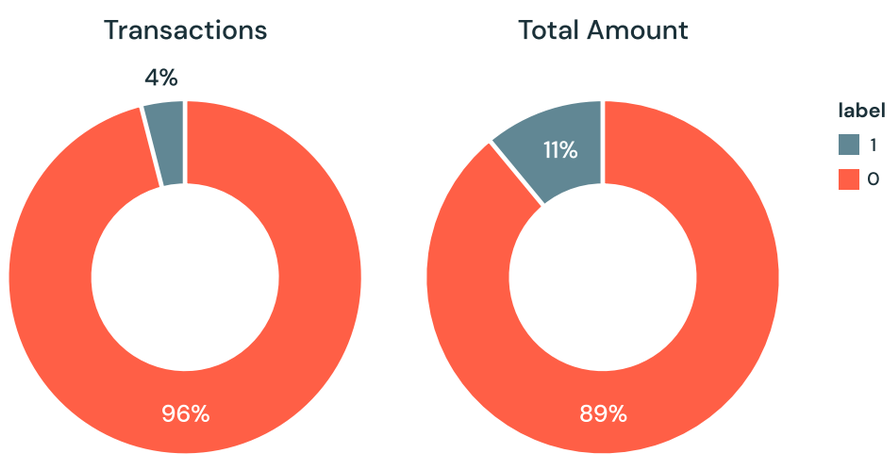
We are not likely to start with a large data set of known fraud cases to train our model. In most practical applications, fraudulent detection patterns are identified by a set of rules established by the domain experts. Here, we create a column called "label" based on these rules.

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

Visualizing data flagged by rules

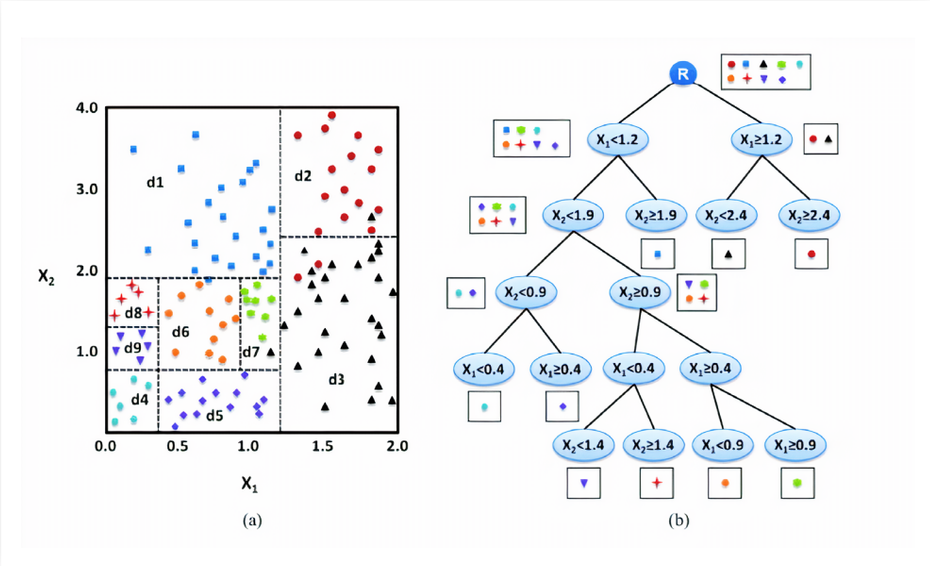
These rules often flag quite a large number of fraudulent cases. Let's visualize the number of flagged transactions. We can see that the rules flag about 4% of the cases and 11% of the total dollar amount as fraudulent.

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



Selecting the appropriate machine learning models

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



Creating the training set

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

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

Creating the ML model pipeline

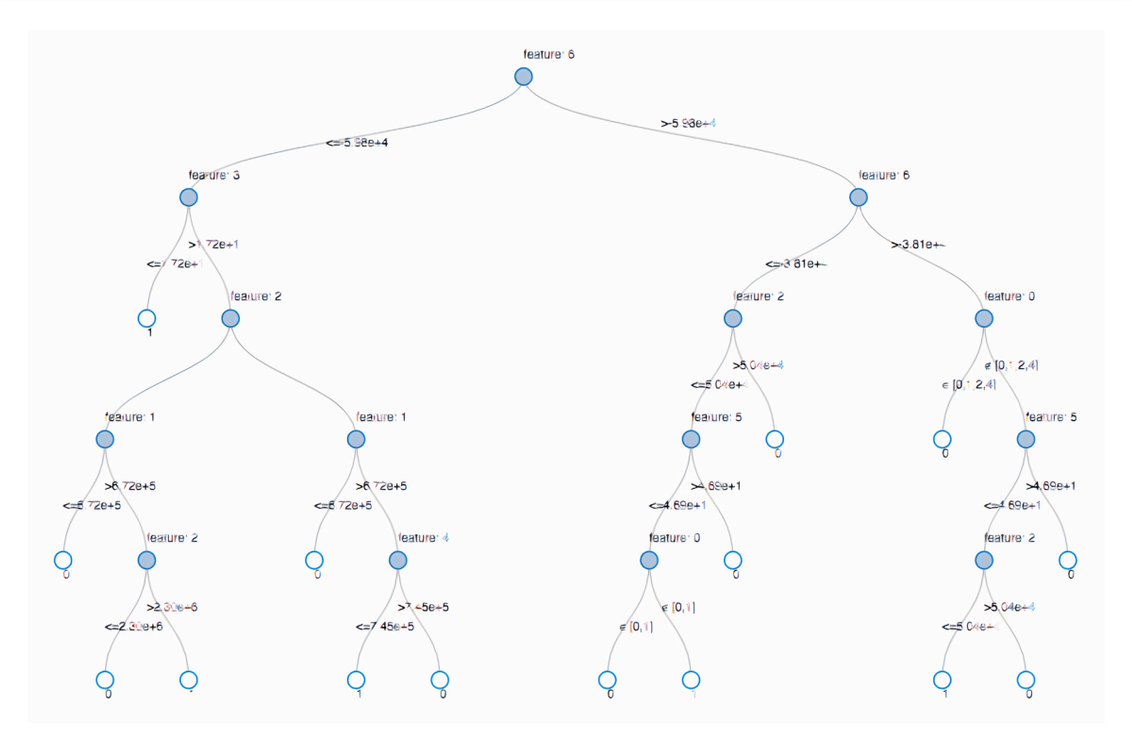
To prepare the data for the model, we must first convert categorical variables to numeric using `.StringIndexer`. We then must assemble all of the features we would like for the model to use. We create a pipeline to contain these feature preparation steps in addition to the decision tree model so that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

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

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

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

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

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

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

Visualizing the model

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

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



Visual representation of the Decision Tree model

Model tuning

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

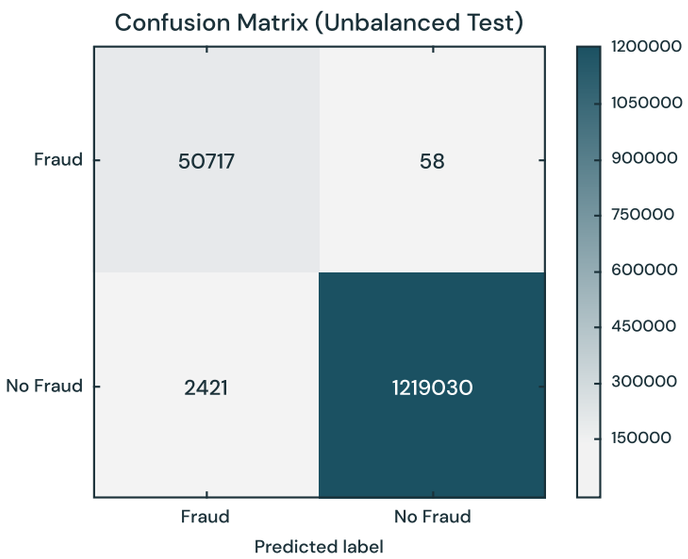
```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

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

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

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

Model performance

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

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

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

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

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

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

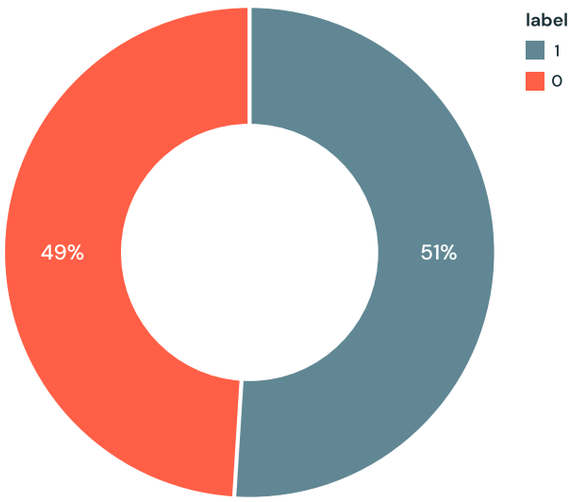
To see how the model misclassified the results, let's use Matplotlib and pandas to visualize our confusion matrix.



Balancing the classes

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

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

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

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

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

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

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



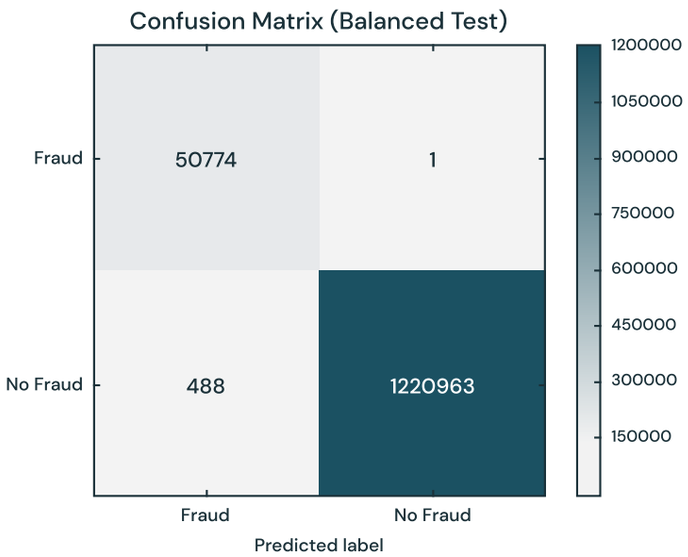
Updating the pipeline

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

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

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

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

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

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

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

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

Review the results

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



Model feedback and using MLflow

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

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

Conclusion

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

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

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

CHAPTER 6:

Automating Digital Pathology Image Analysis With Machine Learning on Databricks

By **Amir Kermany** and **Frank Austin Nothaft**

January 31, 2020

Try this notebook in Databricks →

With technological advancements in imaging and the availability of new efficient computational tools, digital pathology has taken center stage in both research and diagnostic settings. Whole Slide Imaging (WSI) has been at the center of this transformation, enabling us to rapidly digitize pathology slides into high-resolution images. By making slides instantly shareable and analyzable, WSI has already improved reproducibility and enabled enhanced education and remote pathology services.

Today, digitization of entire slides at very high resolution can occur inexpensively in less than a minute. As a result, more and more healthcare and life sciences organizations have acquired massive catalogues of digitized slides. These large data sets can be used to build automated diagnostics with machine learning, which can classify slides — or segments thereof — as expressing a specific phenotype, or directly extract quantitative biomarkers from slides. With the power of machine learning and deep learning, thousands of digital slides can be interpreted in a matter of minutes. This presents a huge opportunity to improve the efficiency and effectiveness of pathology departments, clinicians and researchers to diagnose and treat cancer and infectious diseases.

3 common challenges preventing wider adoption of digital pathology workflows

While many healthcare and life sciences organizations recognize the potential impact of applying artificial intelligence to whole slide images, implementing an automated slide analysis pipeline remains complex. An operational WSI pipeline must be able to routinely handle a high throughput of digitizer slides at a low cost. We see three common challenges preventing organizations from implementing automated digital pathology workflows with support for data science:

1. **Slow and costly data ingest and engineering pipelines:** WSI images are usually very large (typically 0.5–2 GB per slide) and can require extensive image preprocessing.
2. **Trouble scaling deep learning to terabytes of images:** Training a deep learning model across a modestly sized data set with hundreds of WSIs can take days to weeks on a single node. These latences prevent rapid experimentation on large data sets. While latency can be reduced by parallelizing deep learning workloads across multiple nodes, this is an advanced technique that is out of the reach of a typical biological data scientist.
3. **Ensuring reproducibility of the WSI workflow:** When it comes to novel insights based on patient data, it is very important to be able to reproduce results. Current solutions are mostly ad hoc and do not allow efficient ways of keeping track of experiments and versions of the data used during machine learning model training.

In this blog, we discuss how the Databricks Unified Data Analytics Platform can be used to address these challenges and deploy end-to-end scalable deep learning workflows on WSI image data. We will focus on a workflow that trains an image segmentation model that identifies regions of metastases on a slide. In this example, we will use Apache Spark to parallelize data preparation across our collection of images, use pandas UDF to extract features based on pretrained models (transfer learning) across many nodes, and use MLflow to reproducibly track our model training.

End-to-end machine learning on WSI

To demonstrate how to use the Databricks platform to accelerate a WSI data processing pipeline, we will use the Camelyon16 Grand Challenge data set. This is an open-access data set of 400 whole slide images in TIFF format from breast cancer tissues to demonstrate our workflows. A subset of the Camelyon16 data set can be directly accessed from Databricks under /databricks-datasets/med-images/camelyon16/ (AWS | Azure). To train an image classifier to detect regions in a slide that contain cancer metastases, we will run the following three steps, as shown in Figure 1:

1. **Patch Generation:** Using coordinates annotated by a pathologist, we crop slide images into equally sized patches. Each image can generate thousands of patches and is labeled as tumor or normal.
2. **Deep Learning:** We use transfer learning to use a pretrained model to extract features from image patches and then use Apache Spark to train a binary classifier to predict tumor vs. normal patches.
3. **Scoring:** We then use the trained model that is logged using MLflow to project a probability heat map on a given slide.

Similar to the workflow Human Longevity used to preprocess radiology images, we will use Apache Spark to manipulate both our slides and their annotations. For model training, we will start by extracting features using a pretrained InceptionV3 model from Keras. To this end, we leverage pandas UDFs to parallelize feature extraction. For more information on this technique see Featurization for Transfer Learning (AWS | Azure). Note that this technique is not specific to InceptionV3 and can be applied to any other pretrained model.

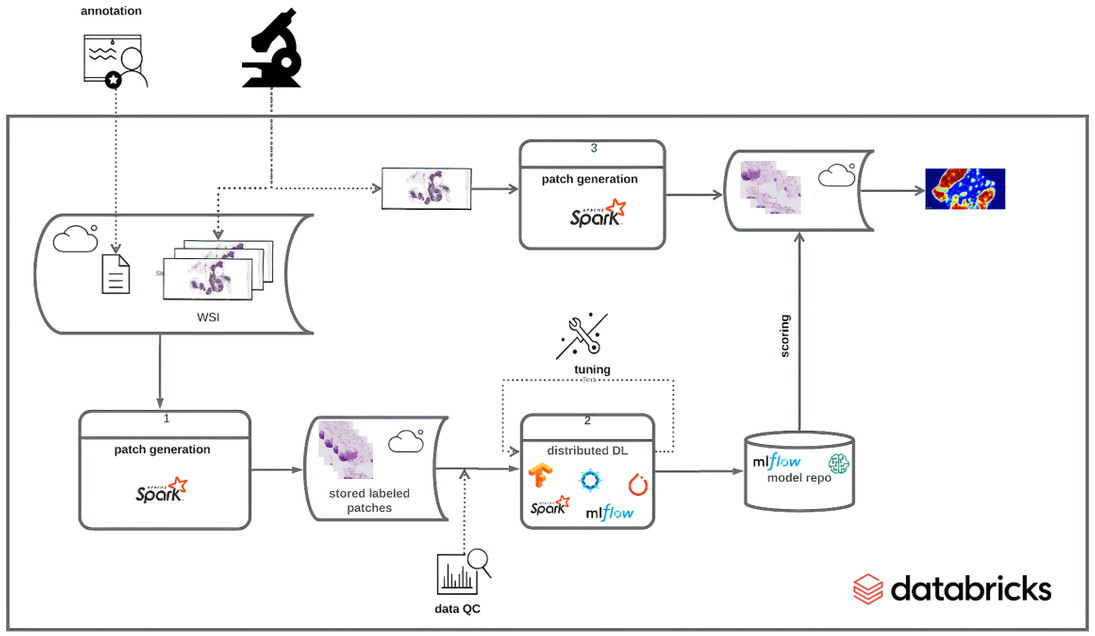


Figure 1: Implementing an end-to-end solution for training and deployment of a DL model based on WSI data

Image preprocessing and ETL

Using open-source tools such as Automated Slide Analysis Platform, pathologists can navigate WSI images at very high resolution and annotate the slide to mark sites that are clinically relevant. The annotations can be saved as an XML file, with the coordinates of the edges of the polygons containing the site and other information, such as zoom level. To train a model that uses the annotations on a set of ground truth slides, we need to load the list of annotated regions per image, join these regions with our images, and excise the annotated region. Once we have completed this process, we can use our image patches for machine learning.

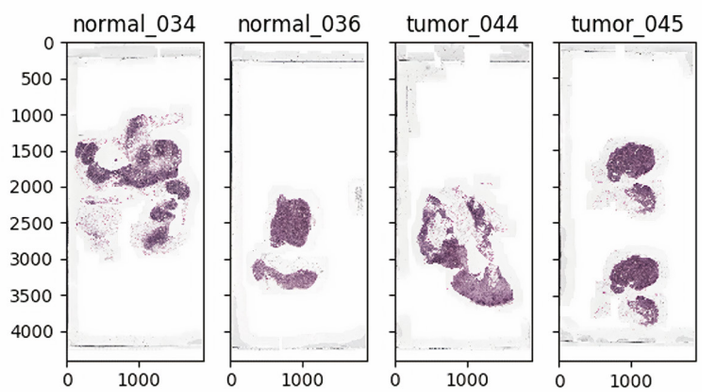


Figure 2: Visualizing WSI images in Databricks notebooks

Although this workflow commonly uses annotations stored in an XML file, for simplicity, we are using the preprocessed annotations made by the Baidu Research team that built the NCRF classifier on the Camelyon16 data set. These annotations are stored as CSV encoded text files, which Apache Spark will load into a DataFrame. In the following notebook cell, we load the annotations for both tumor and normal patches, and assign the label 0 to normal slices and 1 to tumor slices. We then union the coordinates and labels into a single DataFrame.

While many SQL-based systems restrict you to built-in operations, Apache Spark has rich support for user-defined functions (UDFs). UDFs allow you to call a custom Scala, Java, Python or R function on data in any Apache Spark DataFrame. In our workflow, we will define a Python UDF that uses the OpenSlide library to excise a given patch from an image. We define a python function that takes the name of the WSI to be processed, the X and Y coordinates of the patch center, and the label for the patch and creates tile that later will be used for training.

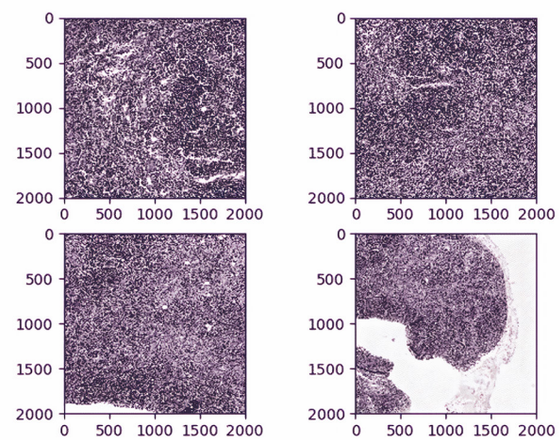


Figure 3: Visualizing patches at different zoom levels

We then use the OpenSlide library to load the images from cloud storage, and to slice out the given coordinate range. While OpenSlide doesn't natively understand how to read data from Amazon S3 or Azure Data Lake Storage, the Databricks File System (DBFS) FUSE layer allows OpenSlide to directly access data stored in these blob stores without any complex code changes. Finally, our function writes the patch back using the DBFS FUSE layer.

It takes approximately 10 minutes for this command to generate ~174,000 patches from the Camelyon16 data set on databricks data sets. Once our command has completed, we can load our patches back up and display them directly in-line in our notebook.

Training a tumor/normal pathology classifier using transfer learning and MLflow

In the previous step, we generated patches and associated metadata, and stored generated image tiles using cloud storage. Now, we are ready to train a binary classifier to predict whether a segment of a slide contains a tumor metastasis. To do this, we will use transfer learning to extract features from each patch using a pretrained deep neural network and then use sparkml for the classification task. This technique frequently outperforms training from scratch for many image processing applications. We will start with the InceptionV3 architecture, using pretrained weights from Keras.

Apache Spark's DataFrames provide a built-in Image schema, and we can directly load all patches into a DataFrame. We then use Pandas UDFs to transform the images into features based on InceptionV3 using Keras. Once we have featurized each image, we use spark.ml to fit a logistic regression between the features and the label for each patch. We log the logistic regression model with MLflow so that we can access the model later for serving.

When running ML workflows on Databricks, users can take advantage of managed MLflow. With every run of the notebook and every training round, MLflow automatically logs parameters, metrics and any specified artifact. In addition, it stores the trained model that can later be used for predicting labels on data. We refer interested readers to these docs for more information on how MLflow can be leveraged to manage a full-cycle of ML workflow on Databricks.

WORKFLOW	TIME
Patch Generation	10 min
Feature Engineering and Training	25 min
Scoring (per single slide)	15 sec

Table 1: Runtime for different steps of the workflow using 2-10 r4.4xlarge workers using Databricks ML Runtime 6.2, on 170,000 patches extracted from slides included in databricks-datasets

Table 1 shows the time spent on different parts of the workflow. We notice that the model training on ~170K samples takes less than 25 minutes with an accuracy of 87%.

Since there can be many more patches in practice, using deep neural networks for classification can significantly improve accuracy. In such cases, we can use distributed training techniques to scale the training process. On the Databricks platform, we have packaged up the HorovodRunner toolkit, which distributes the training task across a large cluster with very minor modifications to your ML code. This blog post provides a great background on how to scale ML workflows on Databricks.

Inference

Now that we have trained the classifier, we will use the classifier to project a heat map of probability of metastasis on a slide. To do so, first we apply a grid over the segment of interest on the slide and then we generate patches — similar to the training process — to get the data into a Spark DataFrame that can be used for prediction. We then use MLflow to load the trained model, which can then be applied as a transformation to the DataFrame, which computes predictions.

To reconstruct the image, we use python's PIL library to modify each tile color according to the probability of containing metastatic sites and patch all tiles together. Figure 4 below shows the result of projecting probabilities on one of the tumor segments. Note that the density of red indicates high probability of metastasis on the slide.

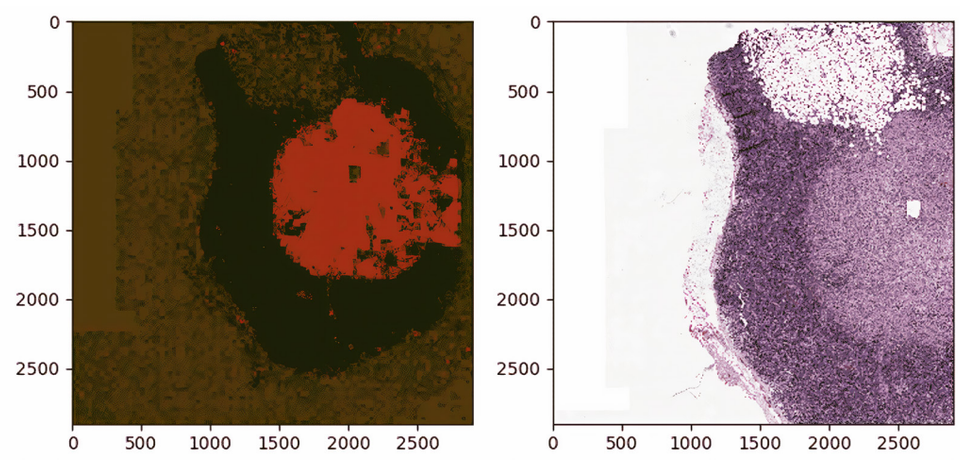


Figure 4: Mapping predictions to a given segment of a WSI

Get started with machine learning on pathology images

In this blog, we showed how Databricks along with Spark SQL, SparkML and MLflow can be used to build a scalable and reproducible framework for machine learning on pathology images. More specifically, we used transfer learning at scale to train a classifier to predict probability that a segment of a slide contains cancer cells, and then used the trained model to detect and map cancerous growths on a given slide.

To get started, sign up for a free Databricks trial and experiment with the WSI Image Segmentation notebook. Visit our healthcare and life sciences pages to learn about our other solutions.

CHAPTER 7:

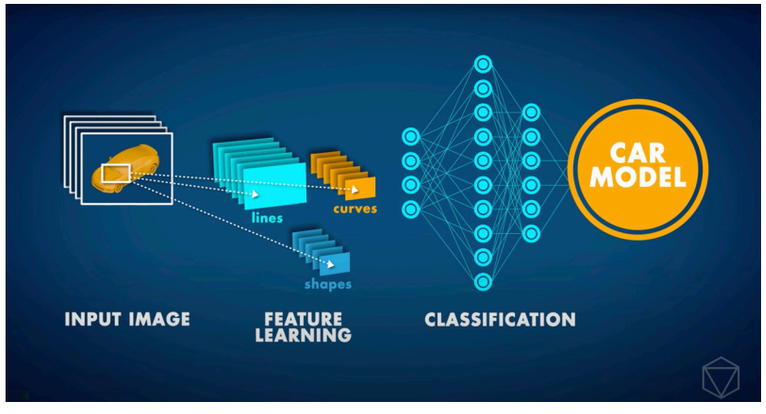
A Convolutional Neural Network Implementation for Car Classification

By **Dr. Evan Eames** and **Henning Kropp**

May 14, 2020

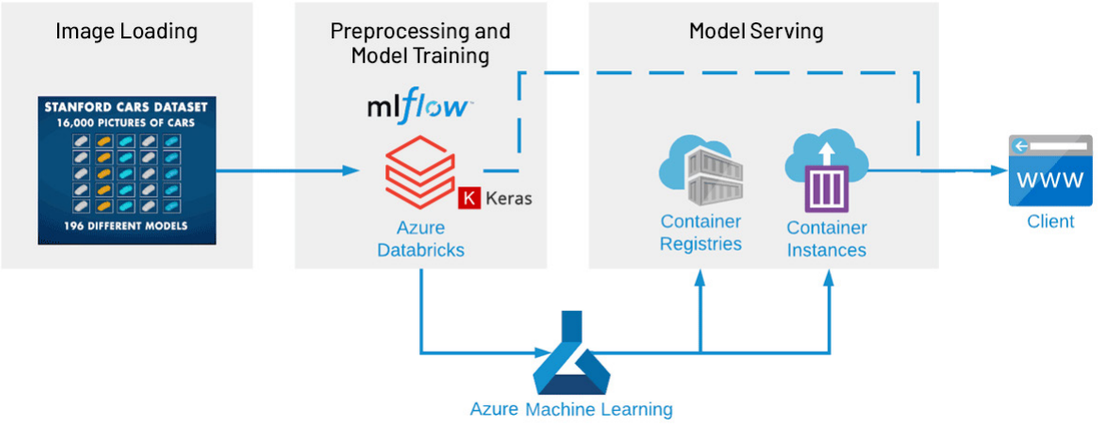
Convolutional Neural Networks (CNN) are state-of-the-art neural network architectures that are primarily used for computer vision tasks. CNN can be applied to a number of different tasks, such as image recognition, object localization, and change detection. Recently, our partner Data Insights received a challenging request from a major car company: Develop a Computer Vision application that could identify the car model in a given image. Considering that different car models can appear quite similar and any car can look very different depending on its surroundings and the angle at which it is photographed, such a task was, until quite recently, simply impossible.

However, starting around 2012, the `Deep Learning Revolution` made it possible to handle such a problem. Instead of being explained the concept of a car, computers could instead repeatedly study pictures and learn such concepts themselves. In the past few years, additional artificial neural network innovations have resulted in AI that can perform image classification tasks with human-level accuracy. Building on such developments, we were able to train a Deep CNN to classify cars by their model. The neural network was trained on the Stanford Cars Data Set, which contains over 16,000 pictures of cars, comprising 196 different models. Over time, we could see the accuracy of predictions begin to improve, as the neural network learned the concept of a car and how to distinguish among different models.



Example artificial neural network, with multiple layers between the input and output layers, where the input is an image and the output is a car model classification.

Together with our partner, we built an end-to-end machine learning pipeline using Apache Spark™ and Koalas for the data preprocessing, Keras with Tensorflow for the model training, MLflow for the tracking of models and results, and Azure ML for the deployment of a REST service. This setup within Azure Databricks is optimized to train networks fast and efficiently, and also helps to try many different CNN configurations much more quickly. Even after only a few practice attempts, the CNN's accuracy reached around 85%.



Setting up an artificial neural network to classify images

In this article, we are outlining some of the main techniques used in getting a neural network up into production. If you'd like to attempt to get the neural network running yourself, the full notebooks with a meticulous step-by-step guide included, can be found below.

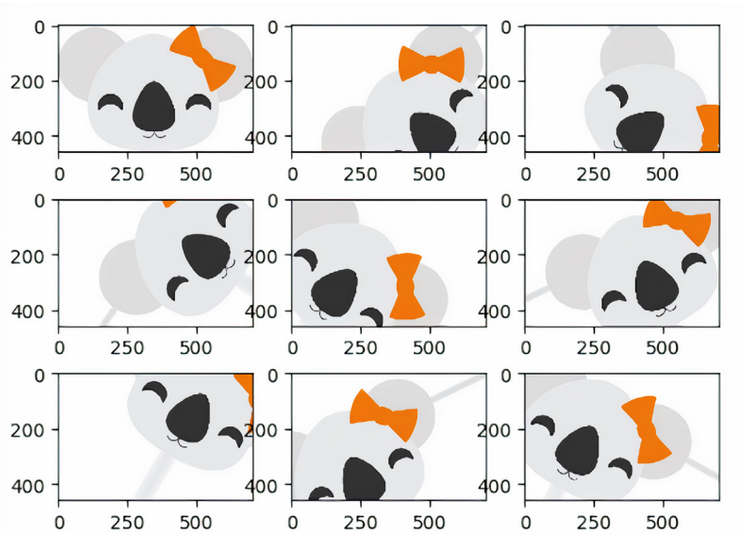
This demo uses the publicly available Stanford Cars Data Set which is one of the more comprehensive public data sets, although a little outdated, so you won't find car models post 2012 (although, once trained, transfer learning could easily allow a new data set to be substituted). The data is provided through an ADLS Gen2 storage account that you can mount to your workspace.

For the first step of data preprocessing, the images are compressed into hdf5 files (one for training and one for testing). This can then be read in by the neural network. This step can be omitted completely, if you like, as the hdf5 files are part of the ADLS Gen2 storage provided as part of these provided notebooks:

- Load Stanford Cars data set into HDF5 files
- Use Koalas for image augmentation
- Train the CNN with Keras
- Deploy model as REST service to Azure ML

Image augmentation with Koalas

The quantity and diversity of data gathered has a large impact on the results one can achieve with deep learning models. Data augmentation is a strategy that can significantly improve learning results without the need to actually collect new data. With different techniques like cropping, padding and horizontal flipping, which are commonly used to train large neural networks, the data sets can be artificially inflated by increasing the number of images for training and testing.



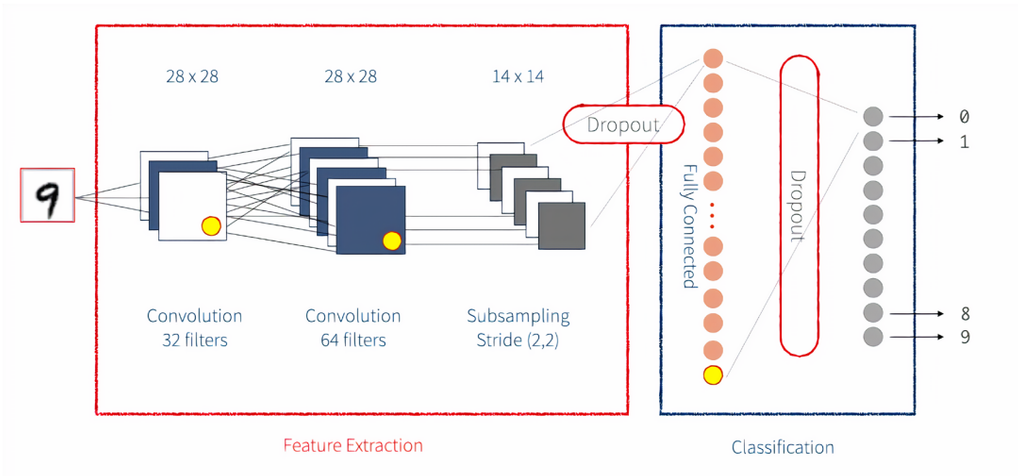
Applying augmentation to a large corpus of training data can be very expensive, especially when comparing the results of different approaches. With Koalas, it becomes easy to try existing frameworks for image augmentation in Python, and scaling the process out on a cluster with multiple nodes using the data science familiar to pandas API.

Coding a ResNet in Keras

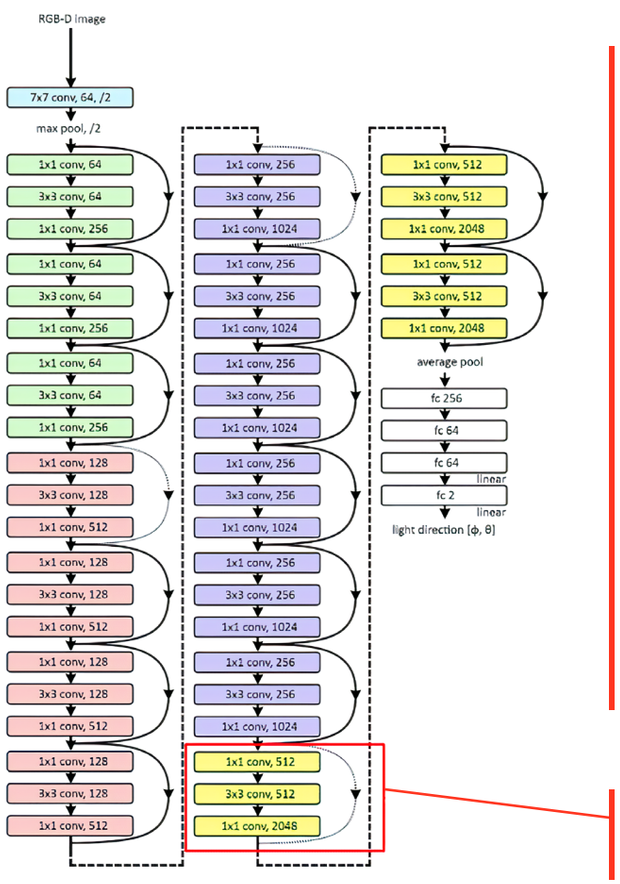
When you break apart a CNN, it comprises different "blocks," with each block simply representing a group of operations to be applied to some input data. These blocks can be broadly categorized into:

- **Identity Block:** A series of operations that keep the shape of the data the same
- **Convolution Block:** A series of operations that reduce the shape of the input data to a smaller shape

A CNN is a series of both Identity Blocks and Convolution Blocks (or ConvBlocks) that reduce an input image to a compact group of numbers. Each of these resulting numbers (if trained correctly) should eventually tell you something useful toward classifying the image. A Residual CNN adds an additional step for each block. The data is saved as a temporary variable before the operations that constitute the block are applied, and then this temporary data is added to the output data. Generally, this additional step is applied to each block. As an example, the below figure demonstrates a simplified CNN for detecting handwritten numbers:



There are many different methods of implementing a Neural Network. One of the more intuitive ways is via Keras. Keras provides a simple front-end library for executing the individual steps that comprise a neural network. Keras can be configured to work with a Tensorflow back-end or a Theano back-end. Here, we will be using a Tensorflow back-end. A Keras network is broken up into multiple layers as seen below. For our network we are also defining our customer implementation of a layer.



The network starts with an input image of size 224 x 224 x 4. Four dimensions represent RGB-D images channels.

The input layer is followed by a convolution layer with 64 kernels with image size of 7 x 7. This layer also uses strides to halve the images size (/2).

The network continues with a max pooling layer which again halves the resolution.

Then, the network contains 48 convolutional layers organized into 16 residual blocks. These residual blocks have an increasing number of kernels.

The convolutional layers are followed by an average pooling and by four fully connected layers with a decreasing number of neurons.

The last layer regresses.

1 of 16 residual blocks, each with 3 convolutional layers.

The Scale layer

For any custom operation that has trainable weights, Keras allows you to implement your own layer. When dealing with huge amounts of image data, one can run into memory issues. Initially, RGB images contain integer data (0-255). When running gradient descent as part of the optimization during backpropagation, one will find that integer gradients do not allow for sufficient accuracy to properly adjust network weights. Therefore, it is necessary to change to float precision. This is where issues can arise. Even when images are scaled down to 224 x 224 x 3, when we use 10,000 training images, we are looking at over 1 billion floating point entries. As opposed to turning an entire data set to float precision, better practice is to use a "Scale layer," which scales the input data one image at a time and only when it is needed. This should be applied after Batch Normalization in the model. The parameters of this Scale layer are also parameters that can be learned through training.

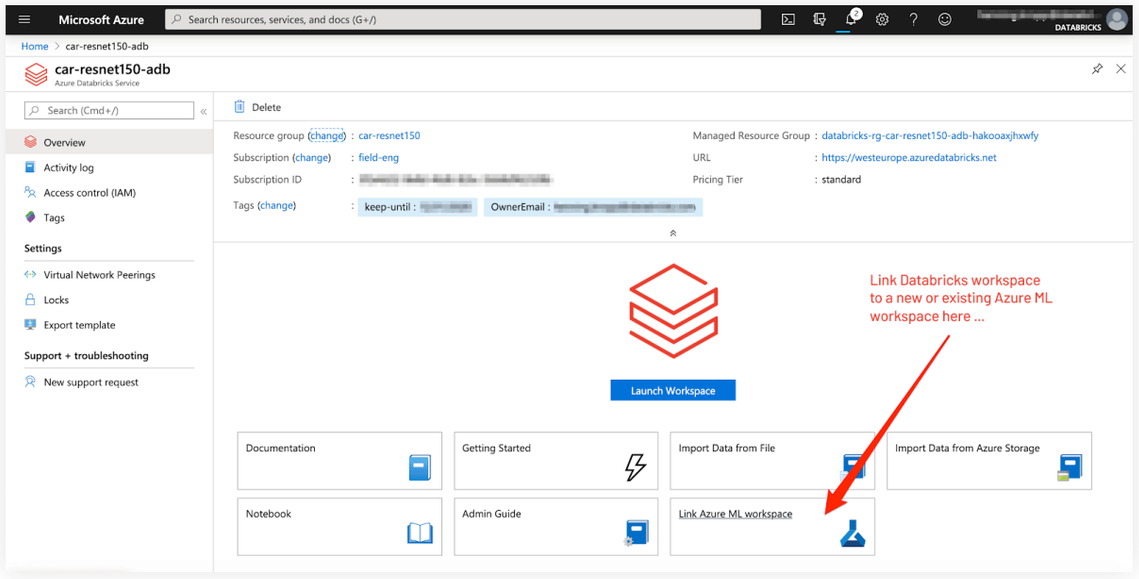
To use this custom layer also during scoring, we have to package the class together with our model. With MLflow we can achieve this with a Keras custom_objects dictionary mapping names (strings) to custom classes or functions associated with the Keras model. MLflow saves these custom layers using CloudPickle and restores them automatically when the model is loaded with `mlflow.keras.load_model()` and `mlflow.pyfunc.load_model()`.

```
mlflow.keras.log_model(model, "model", custom_objects={"Scale": Scale})
```

Tracking results with MLflow and Azure Machine Learning

Machine learning development involves additional complexities beyond software development. That there are a myriad of tools and frameworks makes it hard to track experiments, reproduce results and deploy machine learning models. Together with Azure Machine Learning, one can accelerate and manage the end-to-end machine learning lifecycle using MLflow to reliably build, share and deploy machine learning applications using Azure Databricks.

In order to automatically track results, an existing or new Azure ML workspace can be linked to your Azure Databricks workspace. Additionally, MLflow supports auto-logging for Keras models (mlflow.keras.autolog()), making the experience almost effortless.

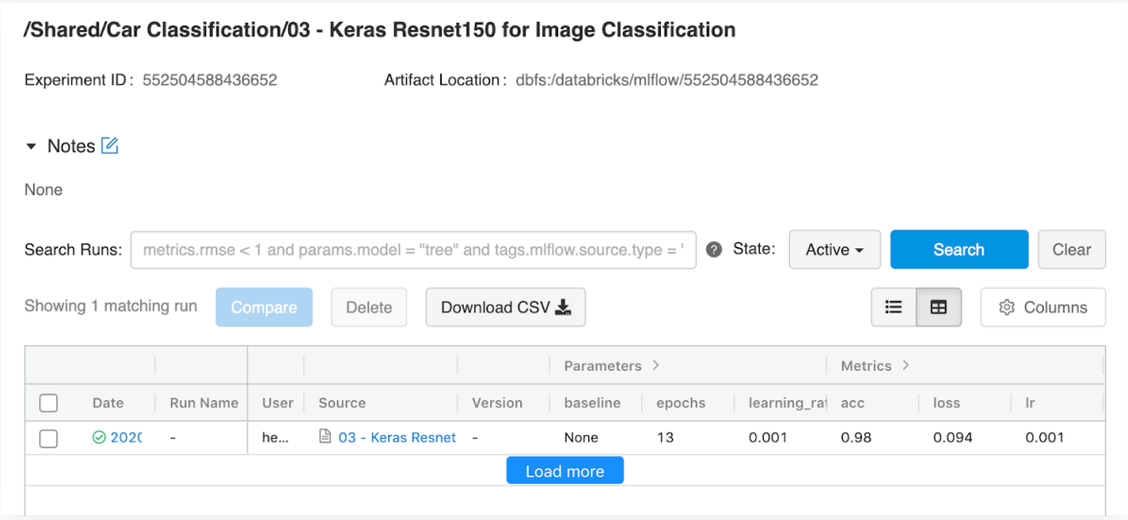


While MLflow's built-in model persistence utilities are convenient for packaging models from various popular ML libraries such as Keras, they do not cover every use case. For example, you may want to use a model from an ML library that is not explicitly supported by MLflow's built-in flavors. Alternatively, you may want to package custom inference code and data to create an MLflow Model. Fortunately, MLflow provides two solutions that can be used to accomplish these tasks: Custom Python Models and Custom Flavors.

In this scenario we want to make sure we can use a model inference engine that supports serving requests from a REST API client. For this we are using a custom model based on the previously built Keras model to accept a JSON DataFrame object that has a Base64-encoded image inside.

```
import mlflow.pyfunc

class AutoResNet150(mlflow.pyfunc.PythonModel):

    def predict_from_picture(self, img_df):
    import cv2 as cv
    import numpy as np
    import base64

    # decoding of base64 encoded image used for transport over http
    img = np.frombuffer(base64.b64decode(img_df[0][0]), dtype=np.uint8)
    img_res = cv.resize(cv.imdecode(img, flags=1), (224, 224), cv.IMREAD_
UNCHANGED)
    rgb_img = np.expand_dims(img_res, 0)

    preds = self.keras_model.predict(rgb_img)
    prob = np.max(preds)

    class_id = np.argmax(preds)
    return {"label": self.class_names[class_id][0][0], "prob": "{:.4}".
format(prob)}

    def load_context(self, context):
    import scipy.io
    import numpy as np
    import h5py
    import keras
    import cloudpickle
    from keras.models import load_model

    self.results = []
    with open(context.artifacts["cars_meta"], "rb") as file:
        # load the car classes file
        cars_meta = scipy.io.loadmat(file)
        self.class_names = cars_meta['class_names']
        self.class_names = np.transpose(self.class_names)

        with open(context.artifacts["scale_layer"], "rb") as file:
            self.scale_layer = cloudpickle.load(file)

        with open(context.artifacts["keras_model"], "rb") as file:
            f = h5py.File(file.name,'r')
            self.keras_model = load_model(f, custom_objects={"Scale":
self.scale_layer})

        def predict(self, context, model_input):
        return self.predict_from_picture(model_input)
```

In the next step, we can use this py_model and deploy it to an Azure Container Instances server, which can be achieved through MLflow's Azure ML integration.



Deploy an image classification model in Azure Container Instances

By now we have a trained machine learning model and have registered a model in our workspace with MLflow in the cloud. As a final step, we would like to deploy the model as a web service on Azure Container Instances.

A web service is an image, in this case a Docker image. It encapsulates the scoring logic and the model itself. In this case, we are using our custom MLflow model representation, which gives us control over how the scoring logic takes in care images from a REST client and how the response is shaped.

```
# Build an Azure ML Container Image for an MLflow model
azure_image, azure_model = mlflow.azureml.build_image(
                    model_uri="{}/py_model"
                         .format(resnet150_latest_run.info.artifact_uri),
                    image_name="car-resnet150",
                    model_name="car-resnet150",
                    workspace=ws,
                    synchronous=True)

webservice_deployment_config = AciWebservice.deploy_configuration()


# defining the container specs
aci_config = AciWebservice.deploy_configuration(cpu_cores=3.0, memory_
gb=12.0)


webservice = Webservice.deploy_from_image(
    image=azure_image,
    workspace=ws,
    name="car-resnet150",
    deployment_config=aci_config,
    overwrite=True)

webservice.wait_for_deployment()
```

Container Instances is a great solution for testing and understanding the workflow. For scalable production deployments, consider using Azure Kubernetes Service. For more information, see how to deploy and where.

Getting started with CNN image classification

This article and the notebooks demonstrate the main techniques used in setting up an end-to-end workflow training and deploying a neural network in production on Azure. The exercises in the linked notebooks will walk you through the required steps of creating this inside your own Azure Databricks environment using tools like Keras, Databricks Koalas, MLflow, and Azure ML.

Developer Resources

- **Notebooks:**
  - Load Stanford Cars data set into HDF5 files
  - Use Koalas for image augmentation
  - Train the CNN with Keras
  - Deploy model as REST service to Azure ML
- **Video:** AI Car Classification With Deep Convolutional Neural Networks on Databricks
- **GitHub:** EvanEames | Cars
- **Slides:** A Convolutional Neural Network Implementation for Car Classification
- **PDF:** Convolutional Neural Network Implementation on Databricks

CHAPTER 8:

Processing Geospatial Data at Scale With Databricks

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

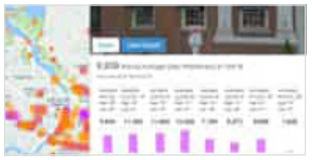
December 5, 2019

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

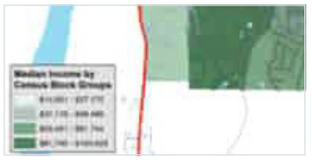
Detect patterns of fraud and collusion (e.g., claims fraud, credit card fraud)

RETAIL



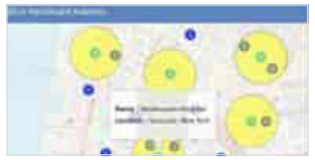
Site selection, urban planning, foot traffic analysis

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

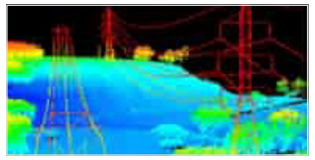
DEFENSE AND INTEL

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 overwhelm traditional storage and processing systems. Customer data has been spilling out of existing vertically scaled geo databases 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, 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 Service (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 NYC 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.

Example: 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.

Example: 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.

Example: 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
[illegible]	[illegible]	[illegible]	[illegible]

Showing the first 1000 rows.

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

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 a 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”))
```

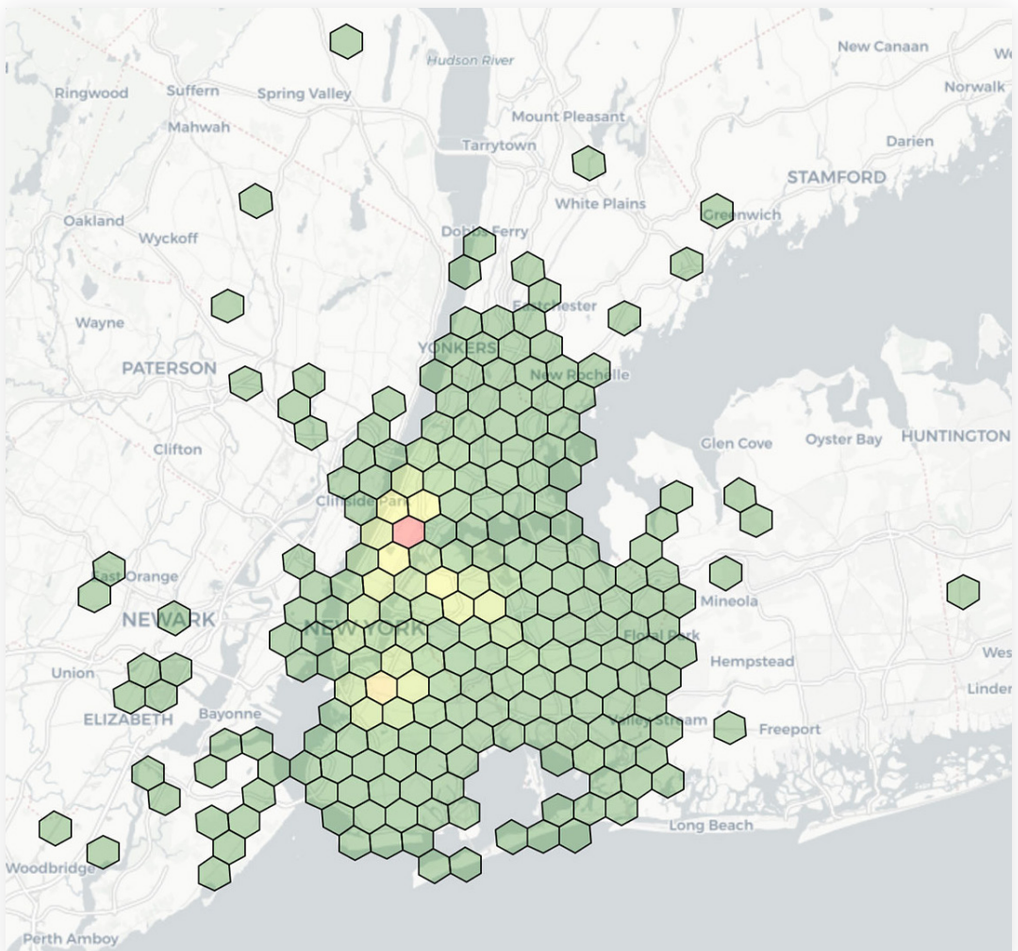
(1) Spark Jobs

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	POINT (-73.95088058740234 40.79750061035156)	2016-06-09 10:02:37	613229520315665311

Showing the first 1000 rows.

Example: 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.



Example: Geospatial visualization of taxi dropoff 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, 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 text 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
```

Example: 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.

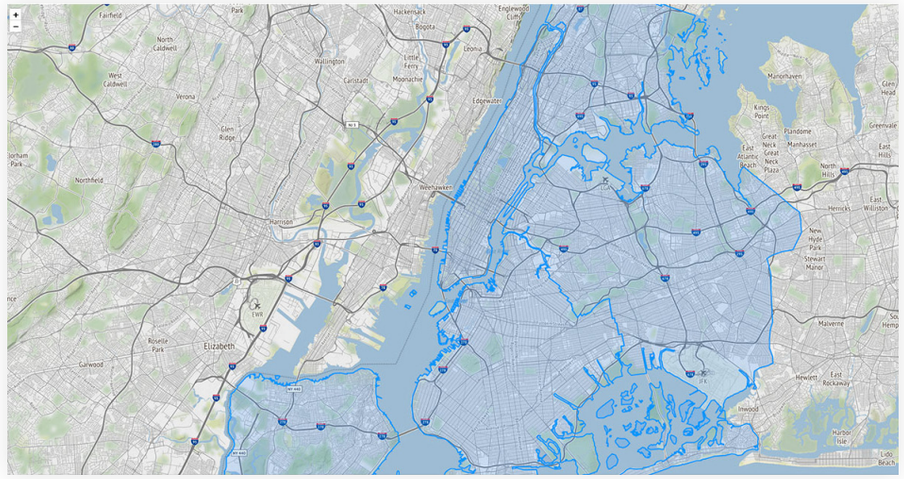
```
%python
from pyspark.sql import functions as F
json_explode_df = ( json_df.select(
 "features",
 "type",
 F.explode(F.col("features.properties")).alias("properties")
).select("*",F.explode(F.col("features.geometry")).alias("geometry")).
drop("features"))

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],[-73.89693872998792,40.79563587285357], [-73.89723603843939,40.79572003753707],[-73.89796839783742,40.795644839161994], [-73.89857332665558,40.7960691402596],[-73.89895261832527,40.796227852579634], [-73.89919434249981,40.79650245601821],[-73.89852052071471,40.796936194189776], [-73.89788253240185,40.79711653214705],[-73.89713149795642,40.79679807772831], [-73.89678526341234,40.79632916648710 5],[-73.89680883223774,40.79580844515979]]], [[[-73.88885148496334,40.798706328958765],[-73.88860021869873,40.798650985918705], [-73.8885856250733,40.798706072297094],[-73.88821348851279,40.798665304638554], [-73.88821415282712,40.798663796217 51],[-73.88825230744402,40.7985771803983], [-73.88837251379924,40.79858745625632],[-73.88839250519693,40.79856629726993],

Example: 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
```



Example: 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.

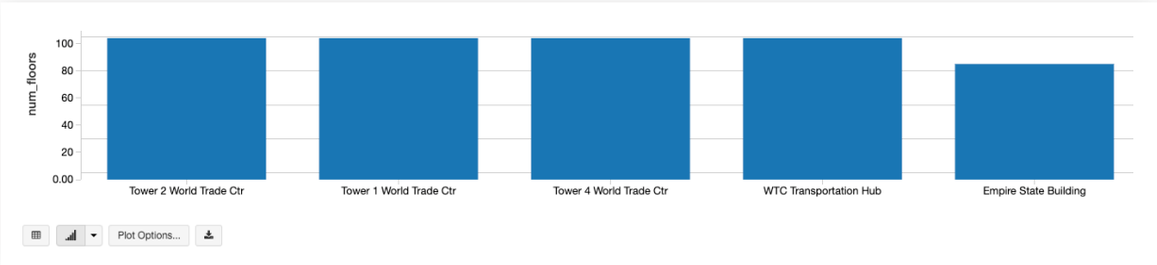
```
%scala
var spatialRDD = new SpatialRDD[Geometry]
spatialRDD = ShapefileReader.readToGeometryRDD(sc, "/ml/blogs/geospatial/
shapefiles/nyc")

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


display(json_explode_df)
```

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
```



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

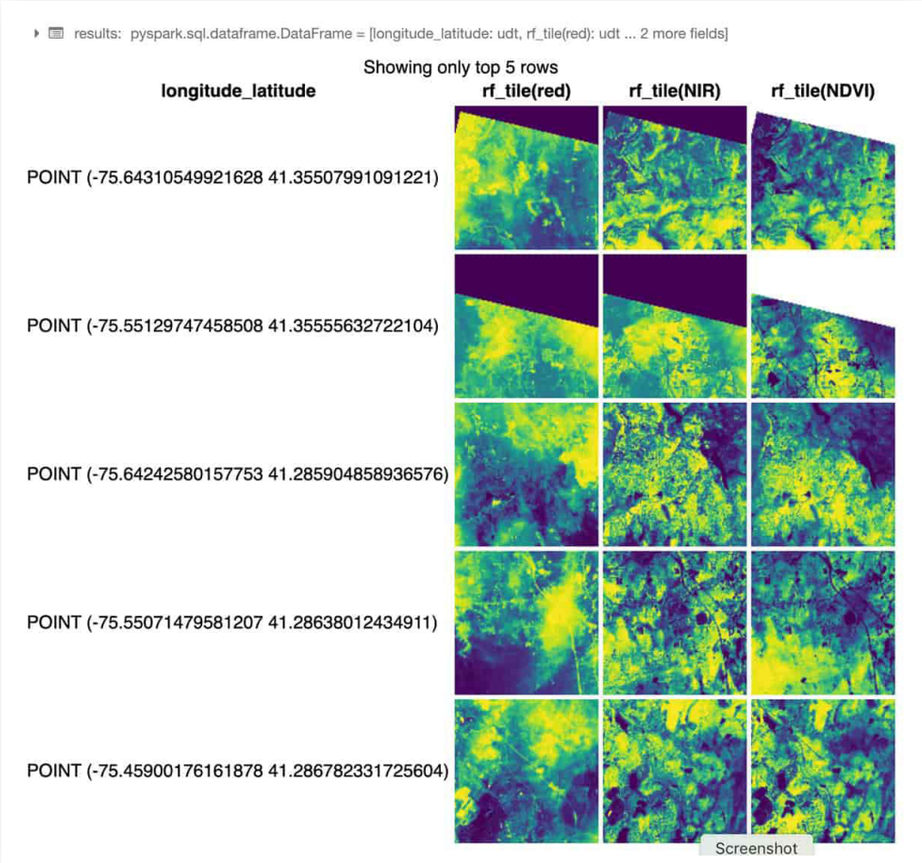
Raster data

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 Pandas DataFrames
bands = [f'B{b}' for b in [4, 5]]
uris = [f'https://landsat-pds.s3.us-west-2.amazonaws.com/c1/L8/014/032/
LC08_L1TP_014032_20190720_20190731_01_T1/LC08_L1TP_014032_20190720_20190
731_01_T1_{b}.TIF' for b in bands]
catalog = ','.join(bands) + '\n' + ','.join(uris)

# read red and NIR bands from Landsat 8 dataset over NYC
rf = spark.read.raster(catalog, bands) \
 .withColumnRenamed('B4', 'red').withColumnRenamed('B5', 'NIR') \
 .withColumn('longitude_latitude', st_reproject(st_centroid(rf_
geometry('red')), rf_crs('red'), lit('EPSG:4326'))) \
 .withColumn('NDVI', rf_normalized_difference('NIR', 'red')) \
 .where(rf_tile_sum('NDVI') > 10000)

results = rf.select('longitude_latitude', rf_tile('red'), rf_
tile('NIR'), rf_tile('NDVI'))
displayHTML(rf_ipython.spark_df_to_html(results))
```

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.



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

Geo databases

Geo databases can be filebased 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, which 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: CUSTOMER CASE STUDIES

Comcast delivers the future of entertainment

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

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

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

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

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

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

Learn more

CHAPTER 9: CUSTOMER CASE STUDIES

Regeneron accelerates drug discovery with genomic sequencing

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

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

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

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

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

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

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

Learn more

CHAPTER 9: 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 as 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 9: 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 9: CUSTOMER CASE STUDIES

Showtime leverages ML to deliver data-driven content programming

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

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

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

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

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

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

Learn more

CHAPTER 9: 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 9: 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 9: 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.

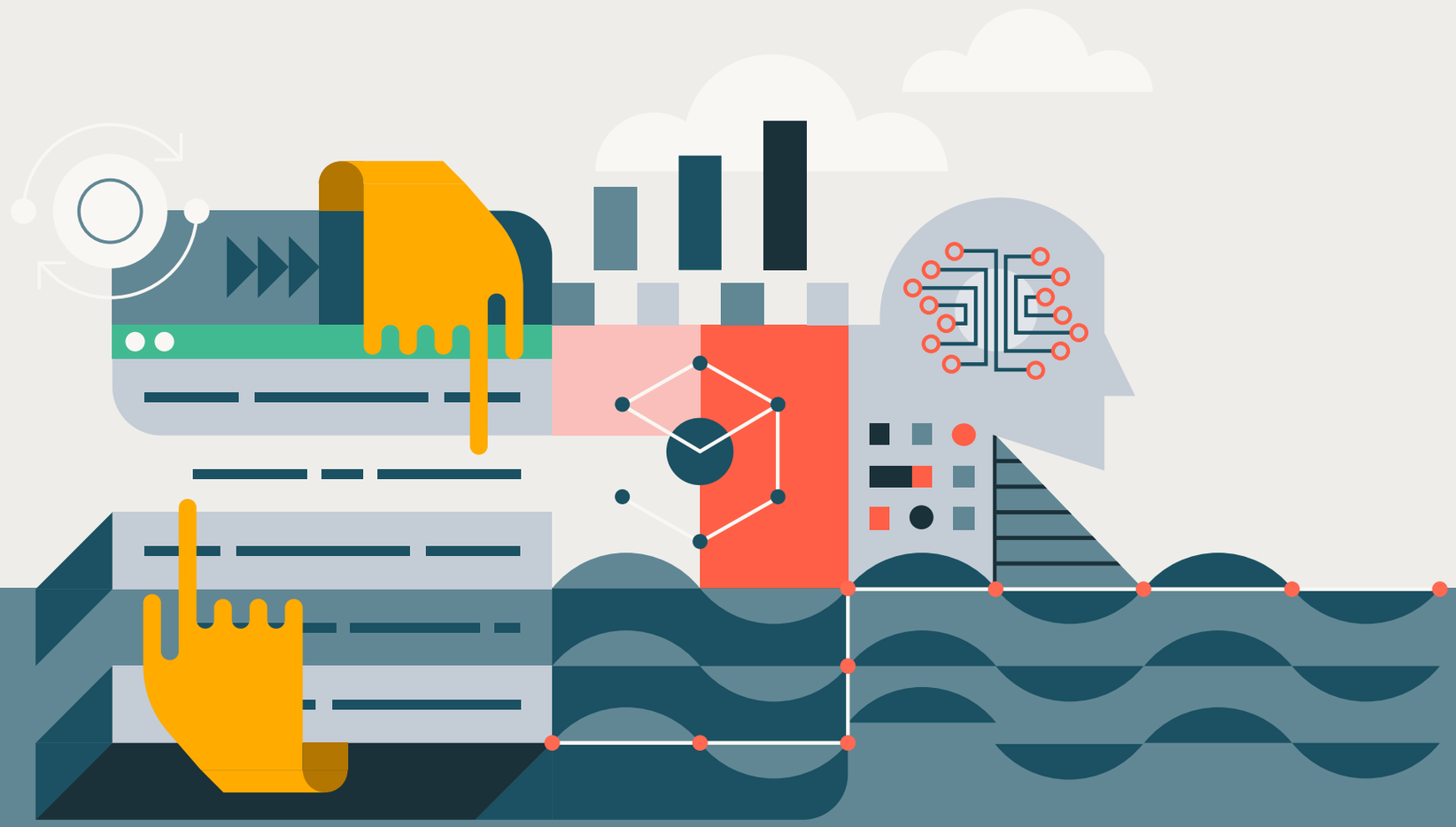
Learn more

About Databricks

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

Schedule a personalized demo

Sign up for a free trial



databricks

© Databricks 2021. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the Apache Software Foundation. Privacy Policy | Terms of Use