# Scaling MLOps to Retrain 50k Weekly Models in Parallel Using UDFs

— Kaleb A. Lowe, PhD —

# Ranks by data.ai: Capturing 360 Mobile Performance

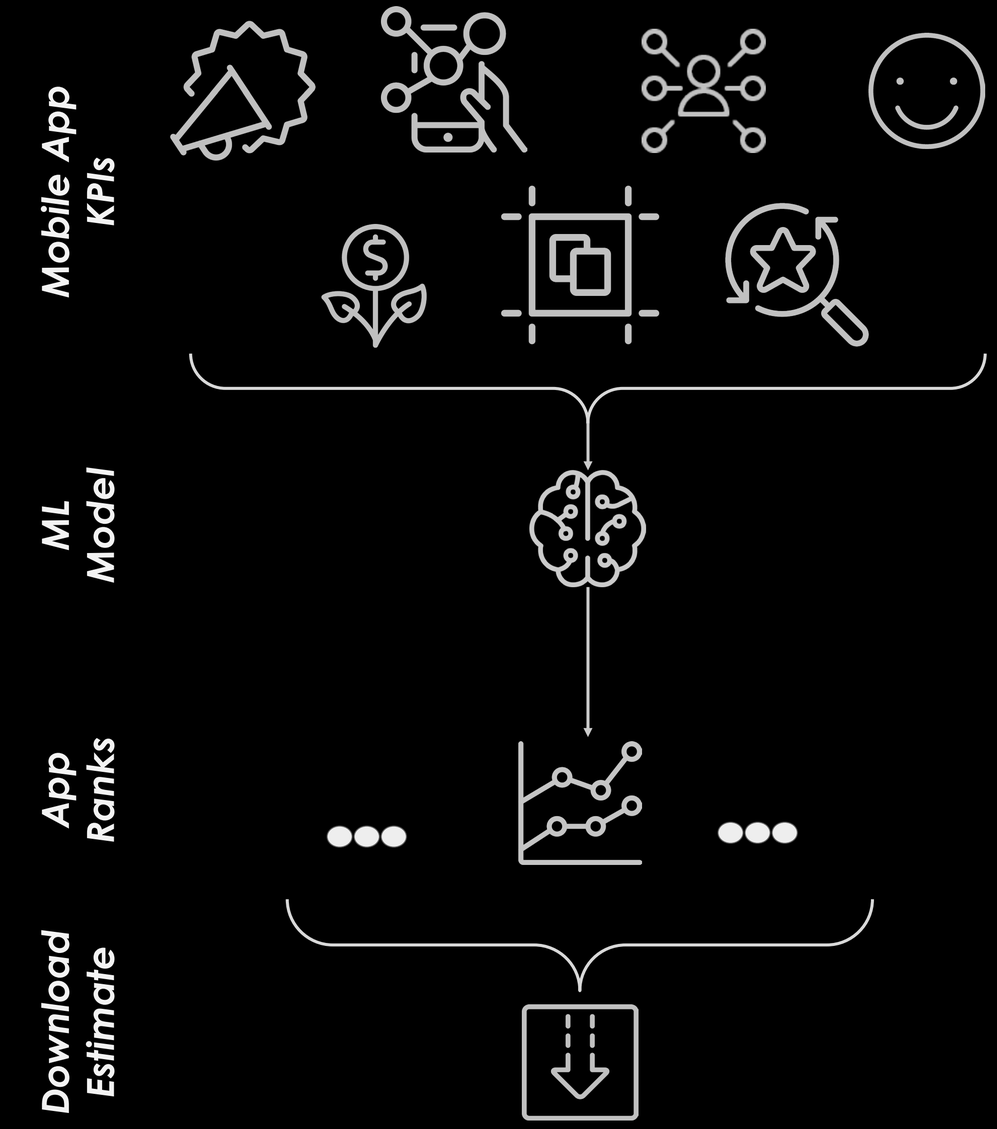
Data.ai is the premier provider of mobile marketplace data and ecosystem insights.

One of data.ai's cornerstone products is our best-in-class downloads estimates.

Downloads estimates are produced using Ranks by data.ai, among other things.

Ranks by data.ai is an ML model that uses our understanding and quantification of the mobile ecosystem to rank app performance.

These ranks and downloads estimates allow our customers to benchmark their performance against their competitors.



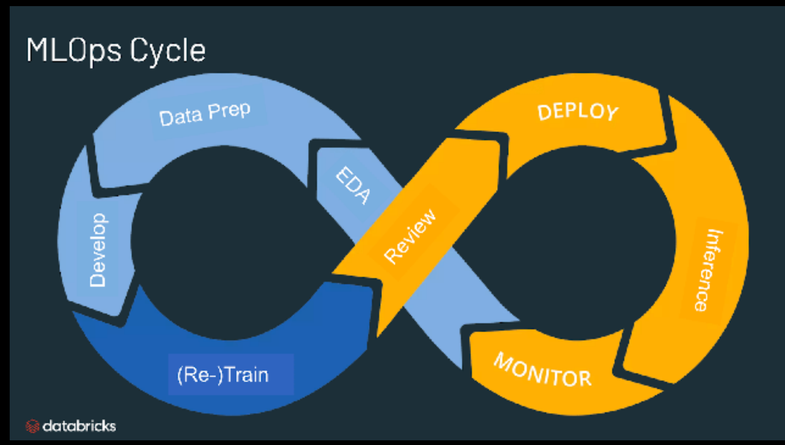
# What is MLOps?

**M**achine **L**earning **Op**eration**s**

According to Databricks: MLOps is "...focused on streamlining the process of taking machine learning models to production, and then maintaining and monitoring them."

MLOps provides the benefits of:
1. Efficiency
2. Risk reduction
3. Scalability

MLOps processes are agnostic to the ML problem or even industry; lessons learned by scaling MLOps are applicable across the board.



# Scaling MLOps at data.ai Presents Challenges

The Ranks by data.ai model has many features, but is itself a simple enough model.

The technical difficulty is in combinatorics: we have to scale to accommodate 175 countries, multiple metrics per country, sub-models, etc.

This product requires managing model training, storage, inference, etc. every week for more than *50 thousand* individual models.

***How do we approach model development, training, and maintenance for 50k models?***

# Agenda

1. Introduction to Pandas UDFs

1. Three tips for scaling MLOps in UDFs
   a. Start small
   b. Wait your turn
   c. Keep things clean

1. Limitations
   a. Batch processing vs real-time
   b. Potential workarounds or extensions

Follow along on Medium!

# Introduction to Pandas UDFs

# Pandas UDFs: The Basics

Pandas UDFs are **U**ser-**D**efined **F**unctions that can be used on a Spark dataframe.

Pandas UDFs vectorize computations and are therefore more efficient.

Regular Spark UDFs operate row-by-row; extra flexibility costs in execution time.

Pandas function APIs are similar to UDFs but operate on full dataframes, allowing for custom aggregations.

# Pandas UDFs: Grouped Average Example

Let's illustrate grouped Pandas functions by creating our own grouped average.

The example is simple, but will show us how we can expand it to ML model training later.

First, let's create a sample dataframe. Let's sample 5 random values and label them 'a', and take another 5 random values, add a constant offset of 3, and label them 'b'.

The built-in grouped average function is a simple groupby/aggregate call.

```
+-----+----------+
|group| avg_value|
+-----+----------+
|    a|0.41722026|
|    b|  3.711982|
+-----+----------+
```

# Pandas UDFs: Grouped Average Example

Instead of the built-in, let's define our own group average UDF.

We just need to do two things: pull the ID of this group and calculate the average value.

- These are returned as a pandas df

To apply this function, we need to specify a return schema using Spark types.

Finally, applying our new function is as simple as using the built-in average:

```
+-----+----------+
|group| avg_value|
+-----+----------+
|    a|0.41722026|
|    b|  3.711982|
+-----+----------+
```

# Building a Model
# Training UDF

# First Tip to Scaling: Start Small

As we've seen before, a UDF is only a function that takes in and returns a pandas dataframe.

We can leverage the fact that this is just a function by building our function against a pandas df.

- We can extract one group from our Spark df and use that to iterate on the UDF.

Now we can start to see how to build out a model training UDF which can ultimately be scaled.

```
def group_average_udf(pdf: pd.DataFrame) -> pd.DataFrame:
  this_group = pdf['group'].values[0]
  group_average = np.mean(pdf['value'])
  return pd.DataFrame({
      'group': [this_group],
      'avg_value': [group_average]
  })
```

```
group_a = test_df.where('group=="a"').toPandas()
a_avg = group_average_udf(group_a)
a_avg.head()
```

| | group | avg_value |
|---|---|---|
| 0 | a | 0.41722 |

# First Tip to Scaling: Start Small

Let’s build a modeling example. The Ranks by data.ai project is based on a random forest, so let’s use a regression example from sklearn.

Let’s also create arbitrary groups: high and low for target values above or below the median.

Finally, let’s extract one group for further testing.

# First Tip to Scaling: Start Small

Now we can write our fitting function. Inside the function, we can simply fit a random forest regressor to the features/target in the df.

Next, we can add MLFlow logging. Most simply, we just start an MLFlow run and then log the model.

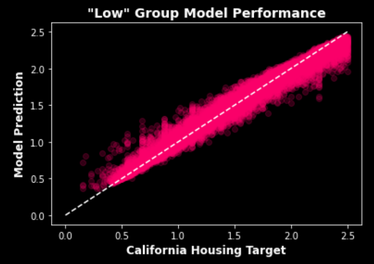
We need to return a pandas df for the UDF to run. The path of the stored model object is a useful return value to keep track of.

Finally, we can test our function on the pandas df for the single group, and see that it works.

We can also load the model object and confirm that it can produce predictions.

```
low_model_path_pdf = fit_california_udf(low_pdf)
low_model_path_pdf
```

| | relative_target | model_path |
|---|---|---|
| 0 | low | runs:/0a6fe30fb20d4b01b4d8cb24caa43947/califor... |

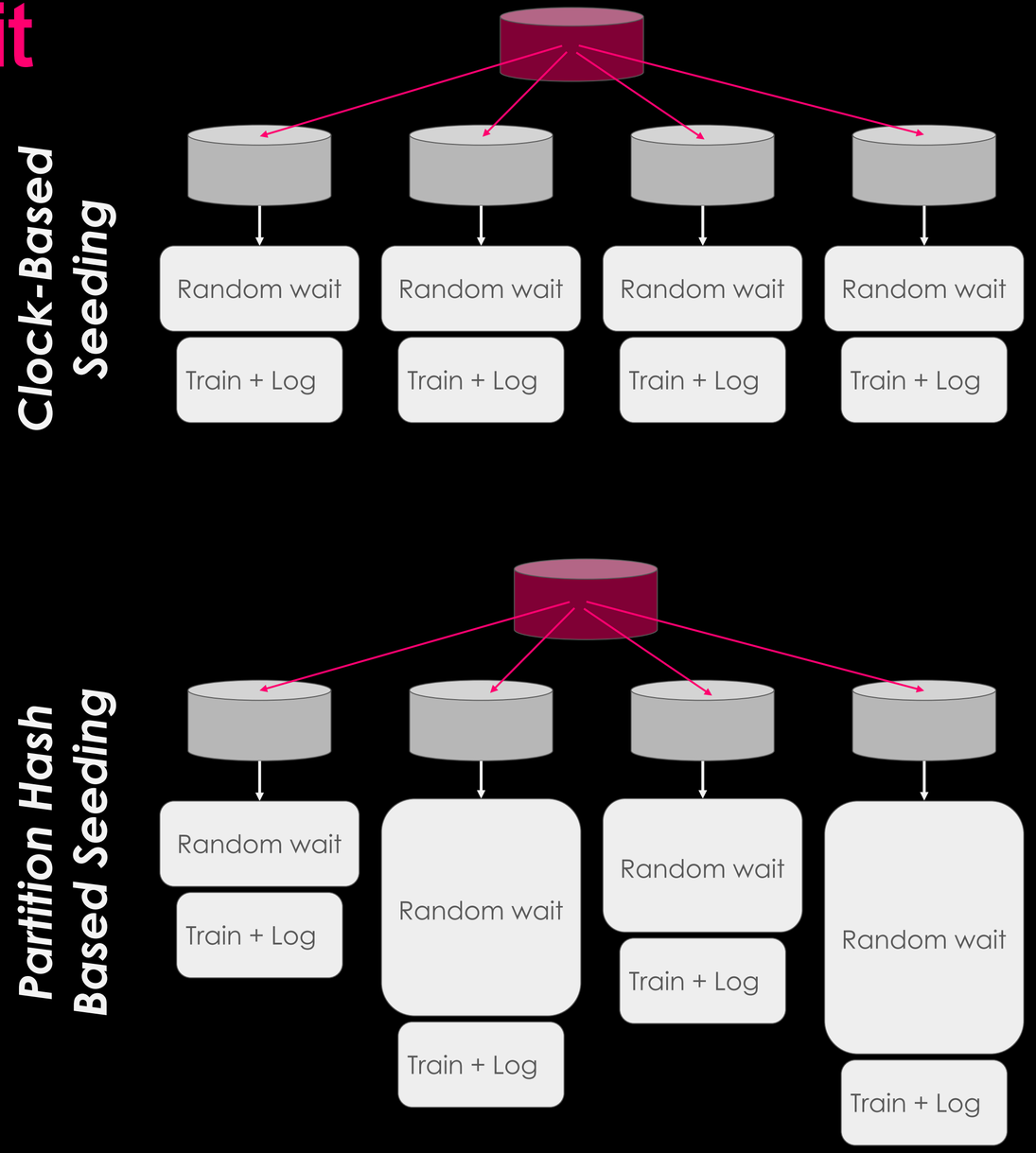


# Second Tip to Scaling: Rate Limit

A sneakier issue with large numbers of groups is the MLFlow API rate limit.

To avoid hitting rate limit errors (429), we need to add a jitter before logging the model so that different groups will be querying at different times.

However, a simple random jitter would not be effective, because many random seeds are determined by the current computer time.



# Second Tip to Scaling: Rate Limit

A better way to handle the rate limit is by using something unique to the group, such as its name.

By using the unique group name (or combination of groups if multiple dimensions), a distinct hash can be used to set the random seed. These are much more likely to be unique and truly avoid rate limit.

```
def fit_california_udf(training_pdf):
  # Initialize and fit
  this_group = training_pdf['relative_target'].values[0]
  this_group_model = RandomForestRegressor()
  this_group_model.fit(training_pdf[features], training_pdf['target'])

  # Log to MLFlow
  run_id = training_pdf['run_id'].values[0]
  experiment_id = training_pdf['experiment_id'].values[0]
  run_params = {'run_id': run_id, 'experiment_id': experiment_id}
  with mlflow.start_run(**run_params, nested=True) as run:




      mlflow.sklearn.log_model(
        this_group_model,
        artifact_path=f"california_model_{this_group}"
      )
      model_path = f"runs:/{run_id}/california_model_{this_group}"

  return pd.DataFrame({
    'relative_target': [this_group],
    'model_path': [model_path]
  })
```

# Interim Summary

1. Pandas UDFs can be used to parallelize arbitrary pandas functions in Spark.

1. UDFs can be developed by using a single group as a pandas df before applying the function to the whole Spark df.

1. Nesting runs packages model artifacts into a single run, useful for organization.

1. MLFlow has a rate limit, but execution can be jittered to avoid getting errors.

1. UDFs can return a path to that group's specific model object in MLFlow.

# Building a Model Inference UDF

# Group Inference UDF

Now that we've stored our grouped models, we need to be able to use them for inference.

We can use our previous lessons to build a UDF for prediction, including using a random jitter to avoid the rate limit error (429).

We can apply this UDF the same way we applied the training, calling it with applyInPandas

- The return schema is constructed from the JSON of the input df schema. This boilerplate is an easy way to add a column to a complex schema without encountering deep/shallow copy issues.

```
def predict_california_udf(features_w_models_pdf):
     # Decode group and model path
   this_group = features_w_models_pdf['relative_target'].values[0]
   this_model = features_w_models_pdf['model_path'].values[0]

   # Add jitter with reset random seed
   random.seed(abs(hash(this_group)) % (10 ** 4))
   rand_wait = random.random()
   time.sleep(rand_wait*5) # Wait random time up to 5 seconds
   # Load model
   this_model = mlflow.sklearn.load_model(this_model)
   features_w_models_pdf['prediction'] = this_model.predict(
      features_w_models_pdf[features]
   )
   return features_w_models_pdf
```

```
# Join model path
df_w_models = california_df.join(
  model_paths_df,
  ['relative_target']
)

# Define return schema
df_json = df_w_models.select('*').schema.jsonValue()
preds_schema = (
  T.StructType()
  .fromJson(df_json)
  .add(T.StructField('prediction', T.FloatType()))
)

# Apply inference
df_w_preds = (
  df_w_models
  .groupBy(['relative_target'])
  .applyInPandas(predict_california_udf, preds_schema)
)
```

# Third Tip to Scaling: Keep Things Clean

The inference UDF does offer opportunities for errors to come in that are hard to diagnose.

One particular error is memory; when we load the model, it stores the model object to the /tmp/ directory on the worker node. If the /tmp/ directory is never cleared then the worker runs out of memory as groups scale up.

We can avoid this memory error by deleting the model objects once we've done inference.

We can do this by specifying where the object is stored when loaded, and then deleting that object after running inference.

```
def predict_california_udf(features_w_models_pdf):
   # Decode group and model path
   this_group = features_w_models_pdf['relative_target'].values[0]
   this_model = features_w_models_pdf['model_path'].values[0]

   # Add jitter with reset random seed
   random.seed(abs(hash(this_group)) % (10 ** 4))
   rand_wait = random.random()
   time.sleep(rand_wait*5) # Wait random time up to 5 seconds



   # Load model
   this_model = mlflow.sklearn.load_model(
      this_model,

   )
   features_w_models_pdf['prediction'] = this_model.predict(
      features_w_models_pdf[features]
   )



   return features_w_models_pdf
```

# Putting It All Together

Now that we've ensured we keep our resources available, let's re-run the inference UDF in Spark and test that the models are in fact separate.

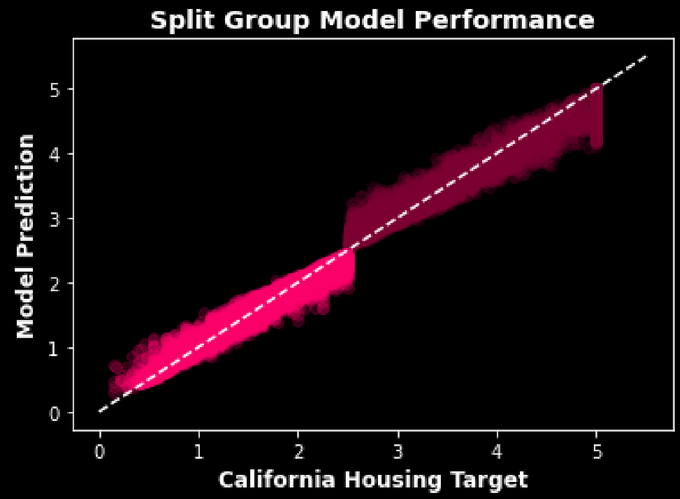
We'll test the model outputs by making a scatter plot, keeping our two groups separate colors.

We can clearly see that the models are doing well, as there is a strong correlation between target and prediction. We also see a clear break in the middle where we split our groups in to two. This trend break confirms the separation of models.

```
df_w_preds = (
  df_w_models
  .groupBy(['relative_target'])
  .applyInPandas(predict_california_udf, preds_schema)
)

grouped_preds_pdf = df_w_preds.toPandas()

plt.scatter(
   grouped_preds_pdf.query('relative_target=="high"')['target'],
   grouped_preds_pdf.query('relative_target=="high"')['prediction']
)
plt.scatter(
   grouped_preds_pdf.query('relative_target=="low"')['target'],
   grouped_preds_pdf.query('relative_target=="low"')['prediction']
)
plt.show()
```



# Interim Summary 2

1. The same parallelization techniques used for model training are also used for inference.

1. In addition to respecting the rate limit, inference UDFs also need to clear out model objects so that worker nodes do not run out of memory from models flooding /tmp/.

1. Adding a column using an existing schema can be done but requires particular boilerplate due to lazy evaluation.

# Limitations and Considerations

# Trivial Considerations

1. There was no cross-validation or other typical model selection process depicted.
   a. These could also be baked into model training, or into a separate UDF

1. We did not register the models
   a. Similarly, model registration could be added to the training procedure
   b. Even a champion-challenger comparison could be done in a registration UDF.

1. What about multiple groups?
   a. No difference in n procedure, just need to incorporate multiple groups into model_paths and jitter values.

# Serious Considerations: Real-Time Inference

1. Long retraining and inference times
   a. Avoiding the rate limit for large combinatorics can be difficult.
   b. Rate-limit avoidance time becomes a limiting factor for real time applications.

1. How to manage 50k model endpoints?
   a. Real-time inference would require some kind of gateway to route traffic to the separate models, based on inputs.

# One Real-Time Suggestion: PyFunc Wrapper

A PyFunc wrapper is a generic MLFlow model object that has greater flexibility than specific flavors (e.g., sklearn, spark, etc.).

PyFunc wrappers have *context*; they can load artifacts into the endpoint.

PyFunc wrapper can be used as a gateway to (1) models in its context or (2) separate model endpoints.

```
class ModelWrapper(mlflow.pyfunc.PythonModel):
   models = {}
   model_groups = ['high', 'low']

   def load_context(self, context):
       import pandas as pd

       self.models = {}
       for model in self.model_groups:
           self.models[model] = mlflow.sklearn.load_model(
               f'models:/california_model_{model}/latest'
           )

   def predict(self, context, input_pdf):
       this_group = input_pdf['relative_target'].values[0]
       this_model = self.models[this_group]
       predictions = this_model.predict(input_pdf)
       return pd.Series(predictions)
```

```
mlflow.pyfunc.log_model(
   artifact_path='california_model_wrapper',
   python_model=ModelWrapper(),
   registered_model_name='registered_california_model'
)
```

# Conclusion

# What We've Learned

1. Introduction to Pandas UDFs

1. Three tips for scaling MLOps in UDFs
   a. Start small
   b. Wait your turn
   c. Keep things clean

1. Limitations
   a. Batch processing vs real-time
   b. Potential workarounds or extensions

Find this on Medium!

Sensor Tower

# data.ai is now a part of Sensor Tower!

The ML techniques presented today are not specific to data.ai, or to Sensor Tower.

These techniques are not even specific to any industry: they can be used for any grouped ML problem needing tracking.

As we push forward on Sensor Tower's mission to measure the world's digital economy, these skills will allow us to efficiently build and scale brand new ML applications, helping to bridge the gap between companies and customers.

# Thank You

data.ai