Apache Beam and Ensemble Modeling: A Winning Combination for Machine Learning
Who is ML6?

Machine Learning services company.

We help our clients build machine learning applications using technologies such as Apache Beam.
Philippe Moussalli
Machine Learning Engineer, ML6
Agenda

● Motivation
  ○ Ensemble Modeling for solving complex use-cases

● Solution
  ○ **Beam RunInference:**
    ■ Seamless integration of ML in a Beam pipeline for semantic enrichment
    ■ Use multiple Runinference transforms for pipelines with multiple ML models

● Example
Motivation

- Semantic Enrichment: ML models provide semantic information.

- Business needs often involve the use of multiple machine learning models, each addressing a specific subtask and contributing unique capabilities.
Semantic Enrichment of Data

- Categorise: Add specific label
- Summarize
- Sentiment Analysis
- Translate
- Extract important keywords
- Image Annotation
- Image Captioning
- Speech Recognition
- .....
Fig.1. Example of a Multi model pipeline, taken from a tutorial on RunInference on Dataflow: Link
Ensemble Modeling: Sequential vs A/B

Sequential Pattern

A/B Pattern
Problem

Seamlessly integrate ML models in a Beam pipeline for semantic enrichment of data.

Business needs require combining multiple ML models. (Ensemble Modeling)

Solution

RunInference API = Inference with ML model in batch and streaming pipelines, without needing lots of boilerplate code.

RunInference API = Using multiple RunInference transforms, build a pipeline that consists of multiple ML models.
RunInference >> Custom DoFn

Seamlessly integrate ML model in a Beam pipeline for semantic enrichment of data.

```python
torch.Tensor([1, 2, 3])
torch.Tensor([4, 5, 6])
torch.Tensor([7, 8, 9])

beam.BatchElements()
```

Custom DoFn

RunInference
RunInference supports popular ML frameworks
How to use RunInference?

```python
from apache_beam.ml.inference.base import RunInference
with pipeline as p:
    predictions = (p | beam.ReadFromSource('a_source')
                   | RunInference(ModelHandler)
                   )
```
from apache_beam.ml.inference.sklearn_inference import SklearnModelHandlerNumpy
from apache_beam.ml.inference.sklearn_inference import SklearnModelHandlerPandas
from apache_beam.ml.inference.pytorch_inference import PytorchModelHandlerTensor
from apache_beam.ml.inference.pytorch_inference import PytorchModelHandlerKeyedTensor

def use_model_handler(client, uri):
    model_handler = SklearnModelHandlerNumpy(model_uri=uri)

    model_handler = PytorchModelHandlerTensor(state_dict_path=uri, model_class=PytorchLinearRegression, model_params={"input_dim": 1, "output_dim": 1})
from apache_beam.ml.inference.base import KeyedModelHandler
keyed_model_handler = \nKeyedModelHandler(PytorchModelHandlerTensor(...))

with pipeline as p:
    data = p | beam.Create(["img1", np.array([[1,2,3],[4,5,6],...])],
                          "img2", np.array([[1,2,3],[4,5,6],...])],
                          "img3", np.array([[1,2,3],[4,5,6],...])])

predictions = data | RunInference(keyed_model_handler)
Example

Image captioning and ranking with Sequential Pattern:

1. **BLIP**: Image Captioning
2. **CLIP**: Ranking captions

Sequential Pattern:

- Input → RunInference → Output → RunInference → Output

Captions:
- a. A cat wearing a hat, with blue background
- b. A cat in a toy hat that looks like a helicopter
- c. A cat wearing a hat with a propeller on top

Ranked Captions:
- 1. A cat wearing a hat with a propeller on top
- 2. A cat in a toy hat that looks like a helicopter
- 3. A cat wearing a hat, with blue background
BLIP: Image Captioning

- Image captioning: "A man and a dog are reading a book together."
- VQA: "What is the dog wearing?"
- Image-Text Retrieval: "The man sitting on a couch is smiling."

Matching score: 0.75

CLIP: Caption Ranking

- Two dogs running across a grassy field
- Whale fin appearing above the surface of the ocean
- Dirt path in the middle of a forest of pine trees

Text Encoder

Image Encoder
ML Inference Pipeline in Beam as a DAG

1. Read
2. Preprocess (BLIP)
3. RunInference (BLIP)
4. PostProcess (BLIP)
5. CoGroupby Key
6. Preprocess (CLIP)
7. RunInference (CLIP)
8. PostProcess (CLIP)
9. Format Captions

[URLs]
ML Inference Pipeline in Beam as a DAG

```python
with beam.Pipeline() as pipeline:
    img_url_pil_img = (pipeline
        | "ReadUrl" >> beam.Create(images_url)
        | "ReadImages" >> beam.Map(read_img_from_url)
    )

    img_url_captions = (img_url_pil_img
        | "BLIPPreprocess" >> beam.MapTuple(lambda img_url, img: (img_url,
            blip_preprocess(img, processor=blip_processor),
        ))
        | "GenerateCaptions" >> RunInference(model_handler=KeyedModelHandler(blip_model_handler),
            inference_args="max_length": 50, "min_length": 10,
            "num_return_sequences": 5, "do_sample": True,
        )
        | "BLIPPostProcess" >> beam.ParDo(beam.ParDo(BLIPPostProcess(processing=blip_processor)))
    )

    img_url_captions_ranking = (img_url_pil_img, img_url_captions)
        | "CreateImageCaptionPair" >> beam.CoGroupByKey()
        | "CLIPPreprocess" >> beam.ParDo(CLIPPreprocess(processing=clip_processor))
        | "CaptionRanking" >> RunInference(model_handler=KeyedModelHandler(clip_model_handler))
        | "CLIPPostProcess" >> beam.ParDo(beam.ParDo(beam.ParDo(CLIPOutput(processing=clip_processor))))
    )

    img_url_captions_ranking | "FormatCaptions" >> beam.ParDo(FormatCaptions(3))
```
Read Images from URLs

def read_img_from_url(img_url: str) -> Tuple[str, Image]:
    image = Image.open(requests.get(img_url, stream=True).raw)
    return img_url, image

with beam.Pipeline() as pipeline:
    img_url_pil_img = (pipeline
        | "ReadUrl" >> beam.Create(images_url)
        | "ReadImages" >> beam.Map(read_img_from_url)
Preprocess Inputs for BLIP

```python
def blip_preprocess(image: Image.Image, processor: BlipProcessor) -> torch.Tensor:
    inputs = processor(images=image, return_tensors="pt")
    return inputs.pixel_values

blip_processor = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-base")

torch.Tensor
```

 `(Img URL, torch.Tensor)`
Inference using BLIP

Input(torch.Tensor) → Prediction(torch.Tensor)

RunInference (BLIP)

(Img URL, torch.Tensor) → (Img URL, RunInference Output)

```
| "GenerateCaptions"
>> RunInference(
    model_handler=blip_model_handler,
    inference_args={
        "max_length": 50,
        "min_length": 10,
        "num_return_sequences": 5,
        "do_sample": True,
    },
)
```
Inference using BLIP

Input(torch.Tensor)  Prediction(torch.Tensor)

(gen_fn = mod_make_tensor_model_fn('generate'))

blip_model_handler = KeyedModelHandler(
    PytorchModelHandlerTensor(
        state_dict_path="/blip_model.pth",
        model_class=BlipForConditionalGeneration,
        model_params={
            "config": AutoConfig.from_pretrained(model_id)
        },
        max_batch_size=1,
        device = "gpu"
        inference_fn=gen_fn))
class BLIPPostprocess(beam.DoFn):
    def __init__(self, processor: BlipProcessor):
        self.processor = processor

    def process(self, element):
        img_url, output = element
        captions = self.processor.batch_decode(output.inference,
                                skip_special_tokens=True)
        yield img_url, captions

"BLIPPostProcess" >> beam.ParDo(BLIPPostprocess(processor=blip_processor))

(Scene image with text: (Img URL, RunInference Output) PostProcess (BLIP) (Img URL, [A cat wearing a hat, with blue background,
A cat in a toy hat that looks like a helicopter,
A cat wearing a hat with a propeller on top])
Grouping Image and BLIP Output

```
(img URL, Image)   (img URL, [Captions])
```

```
CoGroupby Key
```

```
img_urlCaptions_ranking = {
  ("image": img_url_pil_img, "captions": img_urlCaptions)
  | "CreateImageCaptionPair" >> beam.CoGroupByKey()
}
```
Preprocess Inputs for CLIP

```
class CLIPPreprocess(beam.DoFn):
    def __init__(self, processor: CLIPProcessor):
        self._processor = processor

    def process(self, element):
        img_url, grouped_val = element
        pil_img, captions = grouped_val['image'], grouped_val['captions'][0]
        processed_output = self._processor(text=captions,
                                            images=pil_img,
                                            return_tensors="pt",
                                            padding=True)

        yield (img_url, captions), processed_output

clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
```
Inference using CLIP

`class CLIPWrapper(CLIPModel):`

```python
def forward(self, **kwargs: Dict[str, torch.Tensor]):
    # Squeeze because RunInference adds an extra dimension, which is empty.
    kwargs = {key: tensor.squeeze(0) for key, tensor in kwargs.items()}
    output = super().forward(**kwargs)
    logits = output.logits_per_image
    return logits

clip_model_handler = KeyedModelHandler(PytorchModelHandlerKeyedTensor(
    state_dict_path"./clip_model.pth",
    model_class=CLIPWrapper,
    model_params={
        "config": AutoConfig.from_pretrained("openai/clip-vit-base-patch32")
    },
    max_batch_size=1,))

| "CaptionRanking" >> RunInference(model_handler=clip_model_handler)
```
class CLIPPostProcess(beam.DoFn):
    def __init__(self, processor: CLIPProcessor):
        self._processor = processor

    def process(self, element):
        image_url, captions, prediction = element
        prediction_results = prediction.inference
        prediction_probs = prediction_results.softmax(dim=-1).cpu().detach().numpy()
        ranking = np.argsort(-prediction_probs)
        sorted_caption_prob_pair = [(captions[idx], prediction_probs[idx]) for idx in ranking]
        return [(image_url, sorted_caption_prob_pair)]

    "CLIPPostProcess" >> beam.ParDo(CLIPPostProcess(processor=clip_processor))
Printing the results nicely

```
[(Img URL, [(Caption, Probability)])]
```

FormatCaptions

Image: cat_with_hat

Top 3 captions ranked by CLIP:
1: A cat wearing a hat with a propeller on top
   (Caption probability: 0.4338)
2: A cat in a toy hat that looks like a helicopter.
   (Caption probability: 0.3200)
3: A cat wearing a hat, with blue background.
   (Caption probability: 0.1697)
Takeaways

- RunInference transform eliminates the need for extensive boilerplate code in pipelines with machine learning models.

- Multiple RunInference transforms enable complex pipelines with minimal code for multi-ML models.

- Example pipeline can be used for captioning images for finetuning Stable Diffusion.
Code: [GitHub Link]

Tutorial: [Apache Beam Documentation Link]

Slides: [GitHub Link]
QUESTIONS?

Shubham Krishna

shubham-krishna-998922108

shub-kris
Per Entity Training Pipelines in Apache Beam
We are a group of AI and machine learning experts building custom AI solutions.

Amongst our engineers we have several Apache Beam contributors.
Agenda

- Development of ML applications
  - What is training?
  - What is MLOps?
- What does per entity training mean?
  - Training multiple models rather than a single model?
  - Why use a per entity strategy
- Example per entity training pipeline
- Bonus: Using trained models in a RunInference pipeline
What is machine learning model training?
What is machine learning model training?

def contains_firefly():
    ...

Writing logic to detect the Beam macot is almost impossible
What is training a machine learning model?
What is training a machine learning model?
How are machine learning applications built and deployed?
MLOps: Level 0

Data extraction and analysis → Data preparation → Model training → Model evaluation and validation → Trained model → Model serving

Manual experiment steps

Offline data

ML Ops

Model registry

Prediction service

experimentation/development/test
staging/preproduction/production
MLOps: Level 1
What is per entity training?
Example: Building multilingual chatbot

Guten Tag!  안녕하세요!  Bonjour!
What is per entity training?

Multilingual Large Language Model

Dutch Language Model

Spanish Language Model

English Language Model

Italian Language Model
Example: Detect production defects using sensor data
Example: Detect production defects using sensor data

- Sensor 2
- Sensor 4

  - Cap not mounted properly
  - No defect found

- Sensor 1
- Sensor 3
- Sensor 4

  - Component 1 burnt
  - No defect found
Why use a per entity strategy?
Reduce Model Infrastructure Requirements
Faster training & inference

- Dutch Model
- German Model
- Portuguese Model
- Multilingual Large Language Model
Address fairness and bias
Easier to detect problems
Simpler models have the following advantages:

- Faster training & inference
- Easier debugging
- Less powerful hardware required
- Easier to address bias
But there is one big problem: How do I manage the training of all of these models?
Manage training pipelines
The solution? Apache Beam!

- Apache Beam can handle **streaming** and **batch data**
- Apache Beam can easily **prepare data** for training
- Apache Beam can run on different **runners** depending on the model’s **requirements**
- **Abstraction** in ML libraries allows us to train models with few lines of code
Let’s look at an example of a per entity training pipeline
## Predicting incomes per education level

<table>
<thead>
<tr>
<th>Age</th>
<th>Workclass</th>
<th>Education</th>
<th>Marital Status</th>
<th>Occupation</th>
<th>Relationship</th>
<th>Race</th>
<th>Sex</th>
<th>Hours per Week</th>
<th>Native Country</th>
<th>Compensation</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Private</td>
<td>11th</td>
<td>Never-married</td>
<td>Machine-op-inspct</td>
<td>Own-child</td>
<td>Black</td>
<td>Male</td>
<td>40</td>
<td>USA</td>
<td>&lt;=50K.</td>
</tr>
<tr>
<td>38</td>
<td>Private</td>
<td>HS-grad</td>
<td>Married-civ-spouse</td>
<td>Farming-fishing</td>
<td>Husband</td>
<td>White</td>
<td>Male</td>
<td>50</td>
<td>USA</td>
<td>&lt;=50K.</td>
</tr>
<tr>
<td>28</td>
<td>Local-gov</td>
<td>Assoc-acdm</td>
<td>Married-civ-spouse</td>
<td>Protective-serv</td>
<td>Husband</td>
<td>White</td>
<td>Male</td>
<td>40</td>
<td>USA</td>
<td>&gt;50K.</td>
</tr>
<tr>
<td>44</td>
<td>Private</td>
<td>Some-college</td>
<td>Married-civ-spouse</td>
<td>Machine-op-inspct</td>
<td>Husband</td>
<td>Black</td>
<td>Male</td>
<td>40</td>
<td>USA</td>
<td>&gt;50K.</td>
</tr>
<tr>
<td>18</td>
<td>?</td>
<td>Some-college</td>
<td>Never-married</td>
<td>?</td>
<td>Own-child</td>
<td>White</td>
<td>Female</td>
<td>30</td>
<td>USA</td>
<td>&lt;=50K.</td>
</tr>
</tbody>
</table>
Pipeline overview

Load Data → Clean Data → Group per Education Level → Train Models → Save Models
Split data per education level

<table>
<thead>
<tr>
<th>Age</th>
<th>Occupation</th>
<th>Education</th>
</tr>
</thead>
<tbody>
<tr>
<td>29y</td>
<td>Accountant</td>
<td>Bachelor</td>
</tr>
<tr>
<td>31y</td>
<td>Engineer</td>
<td>Master</td>
</tr>
<tr>
<td>54y</td>
<td>Plumber</td>
<td>Bachelor</td>
</tr>
<tr>
<td>37y</td>
<td>Server</td>
<td>High School</td>
</tr>
<tr>
<td>47y</td>
<td>Barista</td>
<td>High School</td>
</tr>
<tr>
<td>22y</td>
<td>Cashier</td>
<td>Bachelor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>29y</th>
<th>...</th>
<th>Accountant</th>
<th>Bachelor</th>
</tr>
</thead>
<tbody>
<tr>
<td>54y</td>
<td>...</td>
<td>Plumber</td>
<td>Bachelor</td>
</tr>
<tr>
<td>22y</td>
<td>...</td>
<td>Cashier</td>
<td>Bachelor</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>31y</td>
<td>...</td>
<td>Engineer</td>
<td>Master</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>37y</td>
<td>...</td>
<td>Server</td>
<td>High School</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>47y</td>
<td>...</td>
<td>Barista</td>
<td>High School</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Train model per dataset

- Model 1
  - 29y, Accountant, Bachelor
  - 54y, Plumber, Bachelor
  - 22y, Cashier, Bachelor

- Model 2
  - 31y, Engineer, Master

- Model 3
  - 37y, Server, High School
  - 47y, Barista, High School
with beam.Pipeline(options=pipeline_options) as pipeline:
    _ = (  
        pipeline | "Read Data" >> beam.io.ReadFromText(known_args.input)  
        | "Split data to make List" >> beam.Map(lambda x: x.split(',','))  
        | "Filter rows" >> beam.Filter(custom_filter)  
        | "Create Key" >> beam.ParDo(CreateKey())  
        | "Group by education" >> beam.GroupByKey()  
        | "Prepare Data" >> beam.ParDo(PrepareDataforTraining())  
        | "Train Model" >> beam.ParDo(TrainModel())  
        | "Save" >> fileio.WriteToFiles(path=known_args.output, 
                                      sink=ModelSink()))
def custom_filter(element):
    return len(element) == 15 and '?' not in element \
    and ' Bachelors' in element or ' Masters' in element \
    or ' Doctorate' in element
class PrepareDataforTraining(beam.DoFn):
    def process(self, element, *args, **kwargs):
        key, values = element

        # Convert to dataframe
        df = pd.DataFrame(values)
        last_ix = len(df.columns) - 1
        X, y = df.drop(last_ix, axis=1), df[last_ix]

        # select categorical and numerical features
        cat_ix = X.select_dtypes(include=['object', 'bool']).columns
        num_ix = X.select_dtypes(include=['int64', 'float64']).columns

        # label encode the target variable to have the classes 0 and 1
        y = LabelEncoder().fit_transform(y)

        yield (X, y, cat_ix, num_ix, key)
class TrainModel(beam.DoFn):

    def process(self, element, *args, **kwargs):
        X, y, cat_ix, num_ix, key = element
        steps = [('c', OneHotEncoder(handle_unknown='ignore'), cat_ix),
                 ('n', MinMaxScaler(), num_ix)]

        # one hot encode categorical, normalize numerical
        ct = ColumnTransformer(steps)

        # wrap the model in a pipeline
        pipeline = Pipeline(steps=[('t', ct), ('m', DecisionTreeClassifier())])
        pipeline.fit(X, y)

        yield (key, pipeline)
class ModelSink(fileio.FileSink):
    def open(self, fh):
        self._fh = fh

    def write(self, record):
        _, trained_model = record
        pickled_model = pickle.dumps(trained_model)
        self._fh.write(pickled_model)

    def flush(self):
        self._fh.flush()
Extending the pipeline

... → Train Models → Calculate Metrics
class EvaluateModel(beam.DoFn):
    def __init__(self, model_uri):
        file = FileSystems.open(model_uri, 'rb')
        self.model = pickle.load(file)

    def process(self, element, *args, **kwargs):
        inputs, labels = element
        predictions = self.model.predict(inputs)
        accuracy = sklearn.metrics.accuracy_score(y_pred=predictions, y_true=labels)
        f1 = sklearn.metrics.f1_score(y_pred=predictions, y_true=labels)
        recall = sklearn.metrics.recall_score(y_pred=predictions, y_true=labels)

        file = FileSystems.open(f'model_uri_metrics', 'web')
        file.writelines([f'accuracy: {accuracy}', f'f1: {f1}', f'recall: {recall}'])
How does this pipeline fit in the MLOps architecture?
Let’s try out our model using the RunInference transform
Bonus: Inference in Apache Beam
Summary

- Apache Beam is more and more becoming technology that can be used in advanced MLOps setups
- Per entity strategy has several advantages
  - Requires less powerful hardware
  - Faster training and inference
  - Easier to address bias
  - Easier to debug
- Apache Beam a perfect candidate for per entity training pipelines thanks to
  - Excellent for data preprocessing and preparation
  - Different runners depending on model requirements
  - Abstraction in ML libraries that make it easy to train a model
QUESTIONS?

https://www.linkedin.com/in/jasper-van-den-bossche/
https://github.com/jaxpr
https://www.ml6.eu/
How many ways can you skin a cat, if the cat is a problem that needs an ML model to solve?

Kerry Donny-Clark
Write your own model handler for RunInference!
Power Realtime Machine Learning Feature Engineering with Managed Beam at LinkedIn

David Shao
& Yanan Hao
Optimizing Machine Learning Workloads on Dataflow

Alex Chan
ML model updates with side inputs in Dataflow streaming pipelines
Use Apache Beam to build Machine Learning Feature System at Affirm
Use Apache Beam To Build Machine Learning Feature System At Affirm

- Hao Xu
01

ABOUT ME

# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>01</th>
<th>BACKGROUND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MLFS</td>
</tr>
<tr>
<td></td>
<td>Stream Platform</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>02</th>
<th>PAIN POINTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slowness</td>
</tr>
<tr>
<td></td>
<td>Learning curves</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>03</th>
<th>SOLUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unified transformation</td>
</tr>
<tr>
<td></td>
<td>OOTB APIs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>04</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td>Dev Velocity</td>
</tr>
</tbody>
</table>
Background

- BNPL
- Machine learning feature store
- Streaming and Batch Compute Platform
The Story of BNPL

Your 3 payments of $50.00

Total of payments $150.00
$50.00 is due next month

Set up automatic payments (optional)
You’ll pay $50.00 on each due date.

Complete your order

Pick a payment plan

- $233.00/monthly
  - APR 0.00%
  - Interest $0.00
  - Total $233.00
  - 3 Months

- $120.00/monthly
  - APR 15.01%
  - Interest $22.06
  - Total $142.16
  - 6 Months

- $62.00/monthly
  - APR 15.01%
  - Interest $42.00
  - Total $104.00
  - 12 Months
If a user failed the third payment, is it likely that they will also fail the fourth one?

Has the user failed to make a loan payment, and if so, have we identified the issue? Should we approve another loan for them?
The payment data was processed in batches, resulting in a delay of a couple of days. Utilizing stream data can help prevent such delays in the future.
Figure 1. A feature store is the interface between feature engineering and model development.
Pain Points
Pain Points

**Development Velocity**
Slow backfilling of stream features. Excessive code required to define a feature.

**Variety**
Inability to join two streams from Kinesis together, which is typically required for stateful processing.

**Visibility**
Lack of registry to quickly lookup data sources, features and metadata.
Solution
Complex of **Backfilling**

Backfilling is the process to backfill a feature data to the historical point in time.
Unified Transformation Interface

class UnifiedTransformer(Transformer[beam.PCollection, beam.PCollection]):
    
    @property
def window(self) -> beam.WindowInto:
        return self._window

    @property
def event_transform(self) -> beam.PTransform:
        return self._event_transform

    @property
def aggregator(self) -> beam.PTransform:
        return self._aggregator

    def run(self, inputs: beam.PCollection) -> beam.PCollection:
        if self.feast_context.runner == Runner.flink:
            if self.window:
                inputs = inputs | self.window
                return (inputs | self.event_transform.with_output_types(Tuple)
                         | self.aggregator.with_output_types(Tuple))
            elif self.feast_context.runner == Runner.spark:
                return (inputs | self.event_transform.with_output_types(Tuple)
                         | self.aggregator.with_output_types(Tuple))
        else:
            raise ValueError("Unsupported runner: {0}".format(self.feast_context.runner))
Unified Transformation Interface

```python
@stream_feature_view(
    entities=[entity_registry['user_ari']],
    ttl=timedelta(days=0),
    schema=[
        Field(name="user_ari", dtype=String),
        Field(name="timestamp", dtype=UnixTimestamp),
        Field(name="latest_payment_fail", dtype=UnixTimestamp),
        Field(name="latest_payment_fail_ach_nsf", dtype=UnixTimestamp),
    ],
    online=True,
    source=user_payment_fails_stream_source,
    timestamp_field="timestamp",
    tags={},
    mode="flink",
)

def user_last_payment_fail(feast_context: FeastContext, inputs: PCollection) -> PCollection:
    transformer = UnifiedTransformer(
        feast_context=feast_context,
        aggregator=LatestFeatureAggregator(feast_context, 'timestamp'),
        event_transform=extract_payment_fail_data,
    )
    return transformer.run(inputs)
```
Outcome
Performance boost

**Backfilling time**
Backfilling time improved by 80%

**Code lines**
Reduced 100+ lines to 20+ lines

**Registry**
200+ data sources
100+ features

The time spent to backfill features for feature
- `time_since_user_checkout`
- `time_since_user_last_payment_failure`
Future improvement

1. OOTB transformation interface
2. Transformation framework
3. Improvement on Beam Spark Runner