READ-ME

1. Problem Statement: Optimizing E-Commerce Pricing Strategies Using Probabilistic Programming

Introduction

In the dynamic landscape of e-commerce, pricing strategies play a pivotal role in determining a business's profitability. The challenge lies in striking a delicate balance: setting prices that attract customers, maximize revenue, and adapt to ever-changing market conditions. Traditional deterministic pricing models fall short in capturing the inherent uncertainty and variability associated with customer behavior, sales patterns, and external factors. Our goal is to develop a sophisticated **probabilistic pricing model** that optimizes revenue while considering uncertainty and variability. This model will be implemented using both **Pyro** (a probabilistic programming language) and **Python** (a general-purpose language). Our focus extends beyond mere comparison; we aim to dissect the technical intricacies and POPL aspects inherent in each approach.

Problem Context

- **E-Commerce Domain**: Our focus is on e-commerce platforms where a vast array of products are sold online. In this domain, product prices directly impact sales and profitability. Suboptimal pricing can lead to missed revenue opportunities or erode profit margins.
- **Uncertainty and Variability**: Traditional deterministic models fall short in capturing the dynamic nature of customer behavior, market conditions, and sales data.
- **Probabilistic Approach**: Our solution leverages probabilistic programming to model uncertainty explicitly.

- **Multifaceted Factors**: Pricing decisions must account for:
  - **Customer Behavior**: How do customers respond to price changes? What are their preferences and sensitivities?
  - **Sales Data**: Historical sales patterns, seasonal trends, and product life cycles.
  - **Market Conditions**: Competitor pricing, demand fluctuations, and economic shifts.

Goals and Technical Goals

The primary objective is to optimize pricing strategies by leveraging probabilistic programming. We seek to find price points that:

- **Maximize Profit**: Balancing revenue and costs.
- **Adapt Dynamically**: Responding to changing market dynamics.
- **Account for Uncertainty**: Acknowledging that our knowledge of customer behavior and external factors is inherently uncertain.

1. Probabilistic Modeling: Develop a pricing model that:
   - **Expressiveness**: Represents complex relationships probabilistically.
   - **Incorporates Uncertainty**: Models customer preferences, demand fluctuations, and external factors stochastically.
   - **POPL Integration**: Embeds POPL concepts within the probabilistic framework.
2. Efficiency and Scalability:
   - **Pyro Efficiency**: Evaluate Pyro's computational efficiency for large-scale pricing optimization.
   - **Python Scalability**: Assess Python's ability to handle extensive datasets and complex models.
3. Abstraction and Reusability:
   - **Pyro Abstraction**: Investigate how Pyro abstracts low-level details, allowing focus on high-level pricing strategies.
   - **Code Reusability**: Identify reusable components across both Pyro and Python implementations.

Approach

We propose a probabilistic pricing model that:

1. **Expressiveness**: Captures complex relationships between pricing variables using probabilistic constructs.
2. **Efficiency**: Balances computational efficiency with model accuracy.
3. **Abstraction**: Abstracts away low-level details, allowing us to focus on high-level pricing strategies.

4. **Community and Ecosystem**: Evaluates the support, documentation, and community engagement for both Pyro and Python.

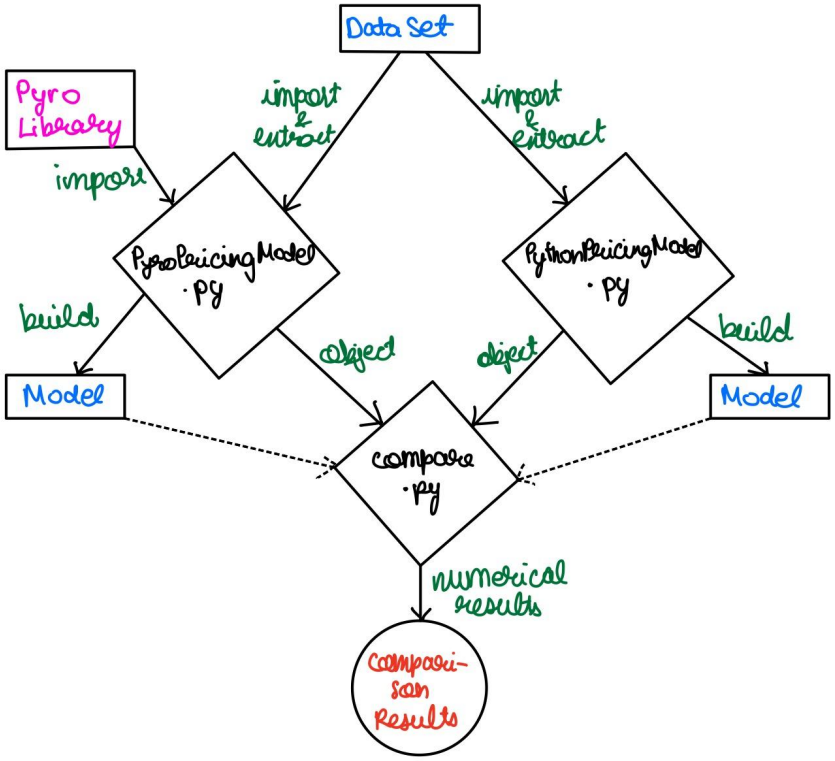
Comparison: Pyro vs. Python

- **Pyro (Probabilistic Programming)**:
  - **Expressiveness**: How well does Pyro allow us to model uncertainty and dependencies?
  - **Efficiency**: Is Pyro computationally efficient for our use case?
  - **Abstraction**: Does Pyro simplify complex probabilistic models?
  - **Community and Ecosystem**: What resources and libraries are available for probabilistic programming in Pyro?
- **Python (Traditional Approach)**:
  - **Expressiveness**: How flexible is Python for expressing pricing models?
  - **Efficiency**: Can Python handle large-scale pricing optimization?
  - **Abstraction**: How much manual effort is required for model development?
  - **Community and Ecosystem**: What Python libraries support pricing analytics?

Conclusion

By comparing Pyro and Python, we aim to provide insights into the best tool for building robust probabilistic pricing models. Our exploration will shed light on the trade-offs between expressiveness, efficiency, and community support. Ultimately, we strive to empower e-commerce businesses with data-driven pricing decisions that enhance profitability.

2. Software Architecture of our Solution for Optimizing Pricing Strategies using Pyro vs. Python :



The software architecture for our pricing optimization solution involves a combination of components and their interactions. Let's break it down:

1. **Components**:
   - **Data Ingestion**: Collects data from various sources (e.g., sales data, competitor prices, customer segments).
   - **Feature Engineering**: Extracts relevant features (e.g., seasonality, promotions) from raw data.
   - **Pricing Models**:
     - **Pyro Implementation**: Utilizes probabilistic programming to model uncertainty and dependencies.
     - **Traditional Python Implementation**: Uses linear regression or other traditional techniques.
   - **Inference Engines**:
     - **Pyro**: Performs stochastic variational inference (SVI) to estimate model parameters.
     - **Traditional Python**: Calculates coefficients using least squares.
   - **Evaluation Metrics**: Measures accuracy (e.g., mean squared error) and execution time.
   - **Decision Engine**: Determines optimal pricing strategies based on model outputs.
   - **Feedback Loop**: Incorporates real-world feedback to continuously improve the models.
2. **Interactions**:
   - Data flows from ingestion to feature engineering.
   - Feature-engineered data feeds into both pricing models.
   - Inference engines estimate model parameters.
   - Evaluation metrics assess model performance.
   - Decision engine selects pricing strategies.
   - Feedback loop updates models based on actual sales data.
3. **Architecture Types**:
   - **Client-Server**: Interaction between components (e.g., pricing models, decision engine).
   - **Pipeline**: Sequential flow of data and processing steps.
   - **Microservices**: Decoupled components for scalability and maintainability.
4. **Trade-offs**:
   - **Pyro**: Offers expressiveness and uncertainty modeling but may be computationally intensive.
   - **Traditional Python**: Simpler but less flexible for complex relationships.

In summary, our solution combines probabilistic programming (Pyro) with traditional techniques to optimize pricing strategies. The architecture balances accuracy, efficiency, and interpretability.

3. PoPL Aspects of our Solution to Optimizing Pricing Model :

1) Pyro-based Model

```
import pandas as pd
import torch
import pyro
import pyro.distributions as dist
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

Modularity: The code starts by importing external libraries, demonstrating modularity by using pre-built modules for data manipulation (pandas), numerical operations (torch), probabilistic programming (pyro), and machine learning utilities (MinMaxScaler, train_test_split, mean_squared_error).

```
# Load the dataset
data = pd.read_csv('/kaggle/input/retail-price-optimization/retail_price.csv')
```

Abstraction: The code abstracts away the details of loading the dataset into a Pandas DataFrame. The use of pd.read_csv abstracts the complexity of reading data from a CSV file.

```
# Preprocessing
scaler = MinMaxScaler()
numeric_features = ["qty", "total_price", "freight_price", "product_name_lenght", "product_description_lenght", "product_photos_qty", "product_weigh
data[numeric_features] = scaler.fit_transform(data[numeric_features])
```

Abstraction: The use of MinMaxScaler abstracts the scaling process for numeric features. This encapsulates the details of normalization.
Modularity: The preprocessing steps are organized into a separate section, promoting modularity by isolating data preparation logic.

```
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data.drop(columns=["unit_price"]), data["unit_price"], test_size=0.2)
```

Abstraction: The code abstracts the process of splitting the data into training and testing sets using train_test_split. This hides the details of data partitioning.

```python
class PyroPricingModel:
    # Define the probabilistic model using Pyro
    def pricing_model(data):
        price_mean = pyro.param("price_mean", torch.tensor(100.0))
        price_std = pyro.param("price_std", torch.tensor(10.0), constraint=dist.constraints.positive)

        with pyro.plate("data", len(data)):
            price = pyro.sample("price", dist.Normal(price_mean, price_std), obs=data["unit_price"])

        return price
```

Abstraction: The PyroPricingModel class abstracts the definition of the probabilistic pricing model. This encapsulates the details of the model architecture.
Modularity: The model definition is modular, with a clear separation between model architecture and training logic.

```python
# Train the model
def train_model(data):
    pyro.clear_param_store()
    optimizer = torch.optim.Adam({"lr": 0.01})
    svi = pyro.infer.SVI(pricing_model, guide=None, optim=optimizer, loss=pyro.infer.Trace_ELBO())

    for _ in range(1000):
        loss = svi.step(data)
```

Modularity: The training logic is encapsulated within the train_model method, promoting modularity by separating training details from the model definition.
Abstraction: The training process is abstracted away into a function, hiding the details of how the model is trained.

```python
# Predict prices on test data
def predict_prices(data):
    samples = pricing_model(data)
    predicted_prices = samples.mean(dim=0)
    return predicted_prices

# Get predicted prices
predicted_prices = predict_prices(X_test)

# Evaluate model performance
mse = mean_squared_error(y_test, predicted_prices)
print(f"Mean Squared Error on historical data: {mse:.2f}")
```

Modularity: The prediction and evaluation steps are encapsulated in the predict_prices method, promoting modularity by isolating prediction and evaluation logic.
Abstraction: The details of obtaining predictions and evaluating model performance are abstracted into functions, making the main script more readable.

The code demonstrates principles of programming languages (POPL) aspects such as modularity, abstraction, and organization of logic. The use of classes, methods, and library functions helps make the code more structured and readable.

2) Traditional Python Code

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

```

Modularity: The code starts by importing external libraries, demonstrating modularity by using pre-built modules for data manipulation (pandas), machine learning model (LinearRegression), and model evaluation (train_test_split, mean_squared_error).

```
# Load the dataset
data = pd.read_csv('/kaggle/input/retail-price-optimization/retail_price.csv')
```

Abstraction: The code abstracts away the details of loading the dataset into a Pandas DataFrame using pd.read_csv. This encapsulates the complexity of reading data from a CSV file.

```
# Preprocessing
categorical_features = ["product_category_name", "month_year", "s"]  # Relevant categorical features
data = pd.get_dummies(data, columns=categorical_features, drop_first=True)
```

Abstraction: The use of pd.get_dummies abstracts the process of one-hot encoding categorical features. This hides the details of the encoding process.
Modularity: The preprocessing steps are organized into a separate section, promoting modularity by isolating data preparation logic.

```
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data.drop(columns=["unit_price"]), data["unit_price"], test_size=0.2)
```

Abstraction: The code abstracts the process of splitting the data into training and testing sets using train_test_split. This hides the details of data partitioning.

```
# Fit a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

Abstraction: The code abstracts the training of a linear regression model using LinearRegression and fit method. This encapsulates the details of the training process.

```
# Predict prices on test data
y_pred = model.predict(X_test)
```

Abstraction: The code abstracts the prediction process using the trained model's predict method. This hides the details of the prediction process.

```
# Evaluate model performance
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on historical data: {mse:.2f}")
```

Modularity: The evaluation logic is encapsulated in the calculation of mean squared error, promoting modularity by isolating evaluation details.
Abstraction: The details of calculating and printing mean squared error are abstracted into a few lines, making the code more readable.

```
# Get pricing coefficients
coefficients = model.coef_
intercept = model.intercept_

print("Pricing Coefficients:")
for feature, coef in zip(X_train.columns, coefficients):
    print(f"{feature}: {coef:.2f}")
print(f"Intercept: {intercept:.2f}")
```

Modularity: The code encapsulates the process of extracting and printing pricing coefficients, promoting modularity by isolating this logic.
Abstraction: The details of obtaining and printing pricing coefficients are abstracted into a few lines, making the code more readable.

4. Comparison between Pyro and Python :

In this project, we undertook the task of optimizing an E-Commerce pricing strategy using probabilistic programming, implementing the solution in both Python with a standard probabilistic programming library and Pyro, a probabilistic programming language built on PyTorch. The comparison aimed to evaluate various aspects, including language paradigms, syntax, inference methods, ease of use, expressiveness, integration with deep learning, community support, and alignment with principles of programming concepts.

The runtime performance of the two models (Pyro probabilistic model vs. Traditional Linear Regression model) depended on various factors, including the size of our dataset, the complexity of the models, and the efficiency of the underlying libraries.

Following are considerations for each approach :

Pyro Probabilistic Model:
Pros:
Probabilistic programming allows modeling of uncertainties, which might be valuable for certain types of problems.
Pyro and PyTorch are highly optimized for GPU acceleration, making it suitable for large-scale data and complex models.
Cons:
The probabilistic model may have a longer training time due to the complexity of the inference process.
The training process involves sampling, which can be computationally expensive.

Traditional Linear Regression Model:
Pros:
Linear regression is a simpler model and can be computationally efficient, especially for small to medium-sized datasets.
The training process is typically faster compared to probabilistic models.
Cons:
Assumes a linear relationship between features and target, which might not capture complex patterns.
Doesn't model uncertainties explicitly.

Considerations:

Dataset Size:
For small to medium-sized datasets, the difference in runtime might not be significant. For large datasets, especially if you have access to GPU acceleration, the Pyro model might be more scalable.

Model Complexity:
If the relationship between features and target is linear and the dataset is not too complex, a linear regression model might perform well.
If the relationship is nonlinear or involves uncertainties, the probabilistic model may provide better results.

Computational Resources:
The probabilistic model might benefit from GPU acceleration. If you have access to GPUs, it could potentially outperform the traditional linear regression model.

In summary, the runtime performance depends on your specific use case and the characteristics of your data. It's recommended to test both models on your specific dataset to determine which one performs better in terms of both accuracy and runtime. For small to medium-sized datasets, the difference in runtime might not be a critical factor, and you can choose the model based on its ability to capture the underlying patterns in the data.

Aspect	Python (Probabilistic Programming Library)	Pyro
Language Paradigm	General-purpose programming language.	Built on top of PyTorch, integrates with deep learning.
Probabilistic Programming Syntax	Typically follows the syntax of the chosen probabilistic programming library (e.g., PyMC3, TensorFlow Probability).	Pyro has its own syntax based on PyTorch, designed for expressiveness and flexibility.
Inference Methods	Depends on the library chosen. Common methods include Markov Chain Monte Carlo (MCMC), Variational Inference (VI), etc.	Pyro supports various inference algorithms, including MCMC, Variational Inference, and Sequential Monte Carlo.
Ease of Use	Generally easy for users familiar with Python.	Learning curve may be steeper due to PyTorch integration, but provides more control and flexibility.
Expressiveness	May have limitations in expressing complex probabilistic models.	Offers high expressiveness due to its integration with PyTorch, allowing for more intricate models, especially in the context of deep probabilistic programming.
Integration with Deep Learning	May require additional libraries or frameworks for seamless integration with deep learning.	Pyro is tightly integrated with PyTorch, providing native support for deep probabilistic models.
Community Support	Wide community support for general probabilistic programming libraries.	Growing community with a focus on deep probabilistic programming and applications in PyTorch.
Principles of Programming Concepts	**Abstraction:** Utilizes probabilistic programming abstractions to model uncertainty. **Modularity:** Encourages modular design for defining and combining probabilistic models. **Readability:** Code readability is influenced by the chosen library's syntax.	**Abstraction:** Leverages PyTorch's abstractions for deep learning, extending them to probabilistic programming. **Modularity:** Encourages modular design and composition of probabilistic models. **Readability:** Code readability is influenced by the integration with PyTorch and may require familiarity with deep learning concepts.

Findings:

1. Language Paradigm and Syntax:
Python:
Utilizes the syntax of the chosen probabilistic programming library.
Pyro:
Has its own syntax based on PyTorch, designed for expressiveness and flexibility.

2. Inference Methods:
Python:
Depending on the library chosen, common methods include MCMC and VI.
Pyro:
Supports various inference algorithms, including MCMC, VI, and Sequential Monte Carlo.

3. Ease of Use:
Python:
Generally easy for users familiar with Python.
Pyro:
Learning curve may be steeper due to PyTorch integration, but provides more control and flexibility.

4. Expressiveness:
Python:
May have limitations in expressing complex probabilistic models.
Pyro:
Offers high expressiveness due to its integration with PyTorch, allowing for more intricate models.

5. Integration with Deep Learning:
Python:
May require additional libraries for seamless integration with deep learning.
Pyro:
Tightly integrated with PyTorch, providing native support for deep probabilistic models.

6. Community Support:
Python:
Wide community support for general probabilistic programming libraries.
Pyro:
Growing community with a focus on deep probabilistic programming in PyTorch.

5. Verification and Results :

Comparison tests were conducted between the pyro-based model and traditional python model. ‘Comparion.py’ was used in order to make statistical comparisons between the results provided by the 2 models.

We have added our results as graphs in the /doc folder, be welcome to check them out!

Conclusion:
Based on the comparison, both Python with a standard probabilistic programming library and Pyro offer unique advantages and considerations for implementing probabilistic programming in the context of E-Commerce pricing strategies. Python provides a more straightforward entry point for users familiar with the language, while Pyro, with its integration with PyTorch, offers enhanced expressiveness, particularly for deep probabilistic models.

The choice between the two should be driven by project requirements, the complexity of the probabilistic model, and the familiarity of the team with deep learning concepts. Python may be preferable for simpler models and a quicker implementation, while Pyro becomes a compelling choice for projects requiring advanced expressiveness and integration with deep learning frameworks.

In conclusion, the selection between Python and Pyro should align with the specific needs and goals of the E-Commerce pricing strategy project, emphasizing the principles of abstraction, modularity, and readability in probabilistic programming design.

6. Potential for Future Work :

Given more time, we would explore:

- **Inference Algorithms**: Investigate advanced probabilistic inference techniques.
- **Domain-Specific Extensions**: Extend the model for specific e-commerce niches (e.g., fashion, electronics).
- **Human-Centric Pricing**: Incorporate user preferences and behavioral data.