```
!pip install hmmlearn
!pip install yfinance
!pip install pgmpy
```

```
Requirement already satisfied: hmmlearn in /usr/local/lib/python3.11/dist-packages (0.3.3)
Requirement already satisfied: numpy>=1.10 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.26.4)
Requirement already satisfied: scikit-learn!=0.22.0,>=0.16 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.6.1)
Requirement already satisfied: scipy>=0.19 in /usr/local/lib/python3.11/dist-packages (from hmmlearn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn!=0.22.0,>=0.16->hmmle
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn!=0.22.0,>=0.16
Requirement already satisfied: yfinance in /usr/local/lib/python3.11/dist-packages (0.2.54)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.11/dist-packages (from yfinance) (2.2.2)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.11/dist-packages (from yfinance) (1.26.4)
Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.11/dist-packages (from yfinance) (2.32.3)
Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/python3.11/dist-packages (from yfinance) (0.0.11)
Requirement already satisfied: platformdirs>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from yfinance) (4.3.6)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.11/dist-packages (from yfinance) (2025.1)
Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/python3.11/dist-packages (from yfinance) (2.4.6)
Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.11/dist-packages (from yfinance) (3.17.9)
Requirement already satisfied: beautifulsoup4>=4.11.1 in /usr/local/lib/python3.11/dist-packages (from yfinance) (4.13.3)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.11/dist-packages (from beautifulsoup4>=4.11.1->yfinance)
Requirement already satisfied: typing-extensions>=4.0.0 in /usr/local/lib/python3.11/dist-packages (from beautifulsoup4>=4.11.1-
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.3.0->yfinance)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.3.0->yfinance) (2025.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinanc
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinance) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinance) (2.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinance) (20
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas>=1.3.0->
Collecting pgmpy
  Downloading pgmpy-0.1.26-py3-none-any.whl.metadata (9.1 kB)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from pgmpy) (3.4.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from pgmpy) (1.26.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from pgmpy) (1.13.1)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (from pgmpy) (1.6.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from pgmpy) (2.2.2)
Requirement already satisfied: pyparsing in /usr/local/lib/python3.11/dist-packages (from pgmpy) (3.2.1)
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (from pgmpy) (2.5.1+cu124)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.11/dist-packages (from pgmpy) (0.14.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from pgmpy) (4.67.1)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (from pgmpy) (1.4.2)
Requirement already satisfied: opt-einsum in /usr/local/lib/python3.11/dist-packages (from pgmpy) (3.4.0)
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (from pgmpy) (2.1.4)
Requirement already satisfied: google-generativeai in /usr/local/lib/python3.11/dist-packages (from pgmpy) (0.8.4)
Requirement already satisfied: google-ai-generativelanguage==0.6.15 in /usr/local/lib/python3.11/dist-packages (from google-gene
Requirement already satisfied: google-api-core in /usr/local/lib/python3.11/dist-packages (from google-generativeai->pgmpy) (2.2
Requirement already satisfied: google-api-python-client in /usr/local/lib/python3.11/dist-packages (from google-generativeai->pg
Requirement already satisfied: google-auth>=2.15.0 in /usr/local/lib/python3.11/dist-packages (from google-generativeai->pgmpy)
Requirement already satisfied: protobuf in /usr/local/lib/python3.11/dist-packages (from google-generativeai->pgmpy) (4.25.6)
Requirement already satisfied: pydantic in /usr/local/lib/python3.11/dist-packages (from google-generativeai->pgmpy) (2.10.6)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.11/dist-packages (from google-generativeai->pgmpy) (4
Requirement already satisfied: proto-plus<2.0.0dev,>=1.22.3 in /usr/local/lib/python3.11/dist-packages (from google-ai-generativ
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas->pgmpy) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->pgmpy) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->pgmpy) (2025.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->pgmpy) (3.5.0
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels->pgmpy) (1.0.1)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.11/dist-packages (from statsmodels->pgmpy) (24.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from torch->pgmpy) (3.17.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch->pgmpy) (3.1.5)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch->pgmpy) (2024.10.0)
```

Step 1

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableEliminatio
```

Training Sets

Training sets are the portion of data we first select to enable the algorithm to learn from the dataset. This allows the model to begin learning the relationships and patterns present in the data, so that the model produced can be used to make predictions on new and unseen data. A Bayesian Belief Network is a graphical model that represents variables and their conditional dependencies through a directed acyclic graph (DAG). The training set is important for the purpose of determining the optimal network structure that best recognizes and models the

relationships and patterns between the data variables. Due to the combinatorial nature of our data structure, we shall employ the hill climbing, heuristic search method to navigate this dataset efficiently (Gámez, 2011). Hill Climbing is a greedy, iterative, optimisation algorithm that is used to solve a diverse set of problems. It begins with an initial, non-optimal solution to a problem, and then attempts to find the optimal solution to a problem by incrementally changing an element of the solution; by aiming to maximize a scoring function, if the change produces a better or more optimal solution, an incremental change is made to the new temporary optimal solution.This process continues until a local optimal solution is found, resulting in a network structure that captures the patterns and relationships present in the dataset (Alvi, 2018). The accuracy of the learned Bayesian Belief Network heavily depends on the quality and representativeness of the training data. An ideal training set is one that captures and is representative of the underlying dataset and its distribution. This is vital for learning an accurate network structure that can produce reliable parameter estimates that can be used going forward for model validation and testing. However, hill climbing algorithms can be computationally expensive and intensive, especially when dealing with large and highly-dimensional datasets. The efficient implementation and use of quality computational resources is necessary to conduct and manage the complexity of the hill climbing search process. Therefore, we understand that the training set is crucial for constructing Bayesian Belief Networks using hill climbing algorithms. It facilitates the discovery of an optimal network structure that accurately represents variable patterns and relationships from the dataset and enables precise estimation of the associated parameters. A well-prepared training set ensures that the resulting BBN can perform reliable inference and support decision-making processes under uncertainty.

1. Definition of a Validation Set A validation set is a subset that is taken from the dataset that will be used to separate the training and the testing test. The approach is used to test the model performance of the model to unseen data to check how the algorithm will perform. The validation os different from the training since it it not used for adjusting the model parameters but instead is used for making decision for the model selection , helping with the fine-tuning the model (hyperparameter tuning) and the stopping criteria which assist with helping the model to be able to generalise well to unseen data before we do the testing (Zhang, X., Zhou ,2016) .
2. Purpose of the Validation Set.

We have many purposes where validation is used. We will mention a few which are the hyperparameter tuning , overfitting and underfitting , model selection and stopping criteria. I will go through each to see how validation is used . Hyperparameter Tuning In the hyperparameter tuning is all about finding the best possible combination of the parameters available to be able to optimize the model performance . Example of the hyper parameter is the learning rates which we can test by different configurations on the validation set one can be able to find the best combination that will generalize to work even on unseen data.

Preventing Overfitting and Underfitting Overfitting occurs when a model has memorized the training data which performs well on the dataset that was used to train it but is unable to generalize to the new data set . That means it will have a good accuracy on the training dataset and a poor performance when we do the validation set . The causes may be the data is not sufficient and that the model includes the noise that is contained in the data. Underfitting happens when a model is too simple to capture the underlying patterns which are found in the data. This means that the model will fail in both the training and validation sets. This means that the model is unable to find the global optimum of the given function which means that it is stuck on the local minimum (Zhou, B and Peng , 2016)

Early Stopping We use the iterative training methods which will help when we can stop. When we have that the validation loss function starts to increase and the training loss function starts to decrease it indicates that we have the potential of overfitting which indicates that we must stop the same applied when we have underfitting (Geubbelmans, Rousseau, A.J , 2023).

Model Selection When we have multiple models trained we can use the validation set to compare their performance. We pick the model which has the best validation performance and is chosen before we do the testing. The validation set ensures that our model is performing well before the final testing . This approach helps the model to perform well when dealing with real world scenarios (Burzykowski, T , 2023).

Step 2

STEP 2 Comparison of the Validation and Testing Sets in Machine Learning We require to properly evaluate the model so that it can generalize well to unseen data. When building the model we divide the dataset into this categories which is , Training Set: For training the model by adjusting its parameters. Validation Set: we use it during training to tune hyperparameters and select the best model. Testing Set: we use it after we are done with training and the validation performance is good so that we can apply the model to real world scenarios and we have confidence it will perform well

2. The Role of the Validation Set The validation set is used during the training phase but is separate from the data the model is directly trained on. It serves several purposes: Hyperparameter tuning: Helps in selecting optimal settings such as learning rate, number of layers, and regularization techniques. Model selection: Allows comparison of multiple models to choose the best-performing one before testing. Preventing overfitting: Ensures that the model is not memorising training data but learning generalizable patterns. Early stopping: Monitors performance and stops training when validation performance starts degrading. When is the Validation Set Used? Repeatedly during training to adjust hyperparameters. Key Differences Between Validation and Testing Sets The validation set and the testing set serve distinct but complementary roles in machine learning. The validation set is used during the training phase to fine-tune hyperparameters, assess different model variations, and prevent overfitting by providing an unbiased evaluation of performance before final model selection. It helps adjust settings such as learning rate, regularization strength, and network architecture. In contrast, the testing set is used only after training is complete to evaluate the final model's generalization to completely unseen data. Unlike the validation set, which can be used multiple times during model development, the test set should be used only once to avoid introducing bias into the evaluation process. While the validation set can indirectly influence the training process by guiding model selection, the

testing set remains untouched until the final evaluation to ensure an objective performance assessment. A poorly allocated validation and testing strategy can lead to overfitting, misleading results, and an unreliable model in real-world applications.

3. The Role of the Testing Set The testing set is used only after training and validation are complete. It provides a final, unbiased measure of how well the model will perform on real-world data. Final performance evaluation: Assesses generalization on completely unseen data. Benchmarking: Compares different models to determine which performs best. Avoids information leakage: Ensures that the test data is not used in any way to tune the model. When is the Testing Set Used? Only once after training and validation are completed. Never used for hyperparameter tuning or model selection.
4. Key Differences Between Validation and Testing Sets
5. Allocating Data for Training, Validation, and Testing The way data is split depends on the size of the dataset: Typical Data Splits 70%-80% for Training 10%-15% for Validation 10%-15% for Testing For example, if we have 100,000 data points, a common split would be: 70,000 samples for training 15,000 samples for validation 15,000 samples for testing Alternative Data Splitting Techniques k-Fold Cross-Validation: Splits the data into k parts and rotates the validation set in each iteration. Stratified Sampling: Ensures each subset has proportional class distribution.

Step 3

Section 4.3 of the dissertation focuses on testing and validating the performance of the constructed probabilistic graphical model for forecasting crude oil prices. Here are the key points likely covered in that section: Testing the Model: This involves simulating trades based on the predictions made by the model. The objective is to assess how well the model performs in real-time trading scenarios.

Stress Testing: The model undergoes stress testing to evaluate its reliability. This means simulating market conditions of economic distress to see how the model behaves under stress and how accurately it reflects price movements during volatile periods.

Performance Metrics: Various metrics would be used to measure the model's accuracy, possibly including prediction error rates, accuracy ratios, and perhaps comparison with baseline models or existing quantitative strategies.

Validation Techniques: There may also be descriptions of cross-validation or out-of-sample tests, which are essential for ensuring that the model does not overfit and can generalize well to unseen data.

Results Interpretation: The outcomes of the validation process are interpreted to provide insights into the model's effectiveness and its practical implications for crude oil trading strategies.

Step 4

```
pip install pgmpy
```

Show hidden output

```
# Create a simple Bayesian Network
model = BayesianNetwork([('Weather', 'Rain'), ('Rain', 'Traffic_Jam')])

# Define the conditional probability distributions (CPDs)
cpd_weather = TabularCPD(variable='Weather', variable_card=2, values=[[0.7], [0.3]])  # P(Weather)
cpd_rain = TabularCPD(variable='Rain', variable_card=2, values=[[0.8, 0.4], [0.2, 0.6]],  # P(Rain | Weather)
                      evidence=['Weather'], evidence_card=[2])
cpd_traffic_jam = TabularCPD(variable='Traffic_Jam', variable_card=2,
                             values=[[0.9, 0.3], [0.1, 0.7]],  # P(Traffic_Jam | Rain)
                             evidence=['Rain'], evidence_card=[2])

# Add the CPDs to the model
model.add_cpds(cpd_weather, cpd_rain, cpd_traffic_jam)

# Perform inference
inference = VariableElimination(model)

# Hill climbing: Optimize CPDs (assuming you're optimizing the P(Rain) table)
def hill_climbing(model, iterations=100):
    # Define the initial state
    best_model = model
    best_score = evaluate_model(model)

    for i in range(iterations):
        new_model = perturb_model(best_model)
        new_score = evaluate_model(new_model)

        if new_score > best_score:
            best_model = new_model
            best_score = new_score
```

```
    return best_model

# Function to evaluate the model (e.g., based on likelihood)
def evaluate_model(model):
    # Here we will simulate the evaluation using some likelihood or objective function
    likelihood = inference.query(variables=['Traffic_Jam'], evidence={'Weather': 1, 'Rain': 0})
    return likelihood.values[1]  # Return probability of Traffic_Jam=1 (traffic jam)

# Function to perturb the model (make small changes to CPDs)
def perturb_model(model):
    # Randomly adjust the CPDs (this is just a simple example of perturbation)
    new_model = model.copy()
    new_cpd_rain = new_model.get_cpds('Rain')

    # Randomly change some probabilities in the CPD (for simplicity, we'll adjust the probabilities)
    new_values = new_cpd_rain.values.copy()

    # Add noise and ensure valid probability distribution
    for i in range(new_values.shape[0]):
        noise = np.random.normal(0, 0.05, new_values.shape[1])
        new_values[i] += noise
        new_values[i] = np.clip(new_values[i], 0, 1)  # Ensure values are within [0, 1]
        new_values[i] = new_values[i] / new_values[i].sum()  # Normalize to ensure it sums to 1

    # Ensure new_values is a 2D list (not a numpy array) for the CPD constructor
    new_values_list = new_values.tolist()

    new_model.remove_cpds('Rain')
    new_cpd_rain = TabularCPD(variable='Rain', variable_card=2,
                              values=new_values_list,
                              evidence=['Weather'], evidence_card=[2])
    new_model.add_cpds(new_cpd_rain)

    return new_model

# Run the hill climbing algorithm
best_model = hill_climbing(model)

# Display the best model
print("Best model after hill climbing:", best_model)
```

```
Best model after hill climbing: BayesianNetwork with 3 nodes and 2 edges
```

The group accuracy for this task was around 79.43%, which helps confirm whether we were able to replicate the results from the paper. While the paper reported a slightly higher accuracy, my result may have been influenced by several factors such as differences in data preprocessing, model tuning, or the computational environment used. Variations in hyperparameters or even random initialization could have contributed to the discrepancy. Despite this, the accuracy achieved is still reasonably close, indicating that the overall methodology may have been effectively replicated

Step 5

Evaluation of Key Contributions To Literature Alvi's paper begins by claiming to substitute the traditional EGARCH-M derived views with Bayesian Model derived views in the Black–Litterman framework. Alvi further references prior work by Beach and Orlov in 2007. Furthermore, the paper converts continuous financial time-series into discrete states (bull, bear, or stagnant regimes) using Hidden Markov Models, and then takes this converted data as inputs for a Bayesian network. Alvi describes this whole design process in Chapter 3 on pages 42 and 43 and then implements the actual process using Python in Chapter 5 from pages 53 to 57. Graphical illustrations of the regime-switching and the Python code excerpts included, show that Hidden Markov Models are able to capture market regimes. Thirdly, Alvi describes a fully implemented trading mechanism that uses the forecasts produced by the Bayesian network. The paper supports this by producing forecasts of the regime state and in turn that informs one's trading actions, whether to buy, sell or do nothing. The plot on page 65 where the model's forecast, the EIA forecast and the actual oil price are compared provide evidence of a trading mechanism that is deployable in commodity markets with independent decision making. In addition to this, the Hill Climbing search helps perform automated structure learning whereby the model is able to learn causal patterns and relationships from the data without relying on extensive expert knowledge apart from the selection of appropriate datasets. Fifthly, Alvi makes use of open-source Python libraries (pgmpy for Bayesian networks and hmms for Hidden Markov Models) to build and test the model. Chapter 4 provides detailed code excerpts and high-level abstractions which help in building a theoretical and practical understanding of the model structures. Moreover, the paper proposes a forecasting method that integrates macroeconomic and geopolitical data to predict crude oil prices, using the power of Bayesian theory. The comparative plot on page 65, displays that the model was able to hedge the risk in the early quarter of 2015, which was caused by geopolitical issues and oil supply crash. The research enables the construction of better models of energy markets by identifying and modeling the patterns and relationships among the key factors that affect oil markets, and this can assist in the policy making process. Finally, the Alvi's integration of

multiple methodologies and findings from various disciplines, that included Bayesian analysis and automated structure learning has created increased alpha for commodity trading. The author's work is a work-in-progress and somewhat accomplishes the proposed results in terms of adapting to market conditions. The author is able to successfully produce a systematic process for feature extraction from noisy data. Moreover, the model exhibits practical deployment potential by successfully hedging risk in the early quarters of 2015, but the author also acknowledges limitations in the model by the mis-timed shorting in a bull run. Although the overall model performance produces mixed results, the strategy's design provides us with an important alternative to traditional methods by embedding fundamental global dynamics into the mode, using Bayesian theory. This contribution is valuable in that it not only serves as a case study in computational finance but also as a reproducible framework for future research Therefore, the author accomplishing some of these proposed results is significant because the true importance lies in moving from theory to practice whereby the model directly enhances trading strategies in real-world settings. This is particularly highlighted in Chapter 4, based on how the constructed Bayesian network can inform responses to economic events, and was able to create an informed prediction on the world oil price that did not veer too far off real world price movements . If these models are refined even further, they could support more effective policy making practices that may be able to prevent recessions or massive financial collapse. Therefore, these contributions are collectively important because they pave the way for more autonomous and robust data-driven decision-making systems in the oil market, with financial implications for both traders and policy makers.

Discussion

Predicting oil prices is a complex procedure that has long relied on traditional financial models, many of which struggle to capture the full range of economic and geopolitical factors that influence market trends and how they are interconnected. In response to this our research paper introduces a new approach that improves upon existing forecasting methods such as GARCH which are based on the assumption that the error variance of the crude oil prices forecast follows the ARMA model, by using Bayesian derived views in a Black-Litterman framework and probabilistic modeling. This new and blended approach breaks down large sets of continuous financial time-series data into discretized regime states (bull, bear and stagnant) using Hidden Markov Models, thus providing a clearer picture of market conditions at different points in time. Traders and analysts are then able to benefit from this by obtaining forecasts that better reflect the ever-shifting and changing oil market regimes, helping them make more informed investment decisions whether the market is rising, falling or stagnant. In addition to this, the new approach is more cost effective because our model is built using open-source Python libraries and automates many of the traditionally manual tasks, whereas a lot of financial modeling tools particularly some of the traditional ones come with heavy computationally intensive requirements making it hard for some research to be conducted. The ability to automate analysis also reduces labor costs, as it minimizes the need for teams of analysts to manually process financial data. This makes the model appealing not only to large investment firms but also to smaller commodity traders who may not have access to expensive forecasting tools. Typically, the traditional forecasting models that have been used prior to this research rely on rigid assumptions about how oil prices behave, such as normality, often failing to account for the non-normal and unpredictable nature of global markets. This research overcomes that limitation by using Bayesian theory, which allows the system to adjust information and predictions as new data becomes available as the model is able to learn from the real-world economic and political events, improving its ability to forecast price movements . Moreover, through the incorporation of Bayesian theory, the model provides insight into how different factors like geopolitical events, economic indicators, and market trends interact to influence oil prices. Moreover, this model is not only subject to historical data; it can integrate real-time updates from economic reports and policy changes. This makes it particularly useful in times of economic uncertainty, where traditional models may lag behind in their predictions. Therefore, the research paper illustrates and provides a new frontier for learning the patterns and relationships of time series data by offering a flexible and efficient way to forecast oil prices. Bayesian theory and concepts are infused throughout the model, because by adapting to new information, we allow our model to have more accurate decision making as it is able to adjust to different market conditions and by reducing the need for constant expert input, we are able to have a model that learns based on the most updated information at hand while being at lower costs due to the use of open-source Python tools.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
from hmmlearn.hmm import GaussianHMM
from pgmpy.models import BayesianNetwork
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.inference import VariableElimination

# ---------------------------
# Step 1: Download Crude Oil Data from Yahoo Finance
# ---------------------------
ticker = "CL=F"
data = yf.download(ticker, start="2018-01-01", end="2023-01-01")
data = data[['Close']].rename(columns={'Close': 'Price'})
data.dropna(inplace=True)

# ---------------------------
# Step 2: Compute Daily Log Returns
# ---------------------------
data['LogReturn'] = np.log(data['Price']).diff()
data.dropna(inplace=True)
```

```
# ---------------------------
# Step 3: Fit a Two-State Gaussian HMM for Regime Detection
# ---------------------------
n_components = 2
hmm_model = GaussianHMM(n_components=n_components, covariance_type="diag", n_iter=1000, random_state=42)
hmm_model.fit(data['LogReturn'].values.reshape(-1, 1))
data['Regime'] = hmm_model.predict(data['LogReturn'].values.reshape(-1, 1))
# Convert regime to string for BN compatibility
data['Regime'] = data['Regime'].astype(str)

# ---------------------------
# Step 4: Discretize Log Returns into Categories
# ---------------------------
bin_labels = ['Low', 'Medium', 'High']
data['ReturnBin'] = pd.qcut(data['LogReturn'], q=3, labels=bin_labels)

# ---------------------------
# Step 5: Build and Fit a Bayesian Network (BN)
# ---------------------------
# Prepare BN data
bn_data = data[['Regime', 'ReturnBin']].copy()

# Instead of using .str.strip(), you can use a list comprehension to strip the column names:
bn_data.columns = [''.join(map(str, col)).strip() for col in bn_data.columns]

# Explicitly convert columns to categorical types with fixed categories.
bn_data['Regime'] = bn_data['Regime'].astype(str)
bn_data['Regime'] = pd.Categorical(bn_data['Regime'], categories=['0', '1'])
bn_data['ReturnBin'] = bn_data['ReturnBin'].astype(str)
bn_data['ReturnBin'] = pd.Categorical(bn_data['ReturnBin'], categories=['Low', 'Medium', 'High'])

# Define the network structure.
model = BayesianNetwork([('Regime', 'ReturnBin')])

# Define state_names explicitly
state_names = {
    'Regime': ['0', '1'],
    'ReturnBin': ['Low', 'Medium', 'High']
}

# Fit the BN model, passing the state_names.
model.fit(bn_data, estimator=MaximumLikelihoodEstimator, state_names=state_names)
infer = VariableElimination(model)

# Compute the mean log return for each ReturnBin (to map discrete predictions to a numerical value).
bin_means = data.groupby('ReturnBin')['LogReturn'].mean().to_dict()

def predict_return_bin(regime):
    # Convert regime to a string if it's a Series:
    if isinstance(regime, pd.Series):
        regime = regime.iloc[0]  # Get the string value of the Series

    query_result = infer.query(variables=['ReturnBin'], evidence={'Regime': regime})
    max_prob = -1
    best_bin = None
    for state, prob in zip(query_result.state_names['ReturnBin'], query_result.values):
        if prob > max_prob:
            max_prob = prob
            best_bin = state
    return best_bin

# ---------------------------
# Step 6: Forecast Next-Day Price Using BN + HMM
# ---------------------------
predicted_log_returns = []
for idx, row in data.iterrows():
    regime = row['Regime']  # Get the individual regime value (string)
    pred_bin = predict_return_bin(regime)  # Pass the individual regime value to function
    if pred_bin is None:
        pred_bin = 'Medium'  # Handle case where pred_bin is None
    predicted_log_returns.append(bin_means[pred_bin])  # Append to the list

data['PredictedLogReturn'] = predicted_log_returns
prev_price = data['Price'].shift(1).values.flatten()
pred_log_return = data['PredictedLogReturn'].values.flatten()
predicted_price = prev_price * np.exp(pred_log_return)
data['PredictedPrice'] = predicted_price
data.dropna(inplace=True)

# ---------------------------
# Step 7: Plot Actual vs. Predicted Prices
# ---------------------------
plt.figure(figsize=(12, 6))
```

```
plt.plot(data.index, data['Price'], label='Actual Price', linewidth=2)
plt.plot(data.index, data['PredictedPrice'], label='Predicted Price', linestyle='--', alpha=0.8)
plt.title('Crude Oil Price Prediction: BN + HMM Forecast vs. Actual')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
[**********100%**********]  1 of 1 completed
<ipython-input-25-5ce4b87006e7>:69: FutureWarning: The default of observed=False is deprecated and will be changed to True in a futu
  bin_means = data.groupby('ReturnBin')['LogReturn'].mean().to_dict()
```