ELECTRICITY PRICE PREDICTION

Using Data Science

Overview:

The energy industry plays a vital role in our modern world, and electricity is a fundamental component of our daily lives. However, the cost of electricity can be highly variable due to factors such as supply and demand, weather conditions, government policies, and market dynamics. Predicting electricity prices is crucial for various stakeholders, including consumers, energy producers, and policymakers, to make informed decisions and manage costs effectively.

Artificial Intelligence (AI) and Python have become powerful tools for forecasting and prediction in numerous domains, including the energy sector. In this project, we will leverage the capabilities of AI and Python to develop a robust electricity price prediction system. By doing so, we aim to provide a valuable resource for both businesses and individuals, allowing them to anticipate and plan for fluctuations in electricity prices.

In this project, we will outline the key steps involved in building an electricity price prediction model using AI and Python. We will discuss the data collection, preprocessing, feature engineering, model selection, and evaluation methods. Additionally, we will explore various AI techniques, such as time series analysis, machine learning, and deep learning, to develop a comprehensive and accurate prediction system.

The project will empower users to:

1. Gain insights into the factors affecting electricity prices.
2. Develop a data-driven understanding of historical price trends.
3. Implement machine learning and AI algorithms for price forecasting.
4. Evaluate and fine-tune the predictive models to improve accuracy.
5. Use the Python programming language to build the prediction system.

By the end of this project, we will have a foundational understanding of how AI and Python can be applied to real-world problems, with a focus on predicting electricity prices. This knowledge can be extended to other areas of predictive analytics, offering valuable insights and decision-making tools in an increasingly data-driven world.

STEPS INVOLVED:

1. Data Collection:

We sourced historical electricity price data from a government energy database. This dataset included daily price records, along with features like demand, generation sources, and weather conditions. The data was collected through web scraping and API calls.

2. Data Preprocessing:

Data cleaning involved identifying and removing missing values. Outliers were detected using statistical methods and then addressed, either by removal or transformation. Data inconsistencies were corrected.

To ensure the data was on a common scale, we applied Min-Max scaling to numerical features.

3. Feature Engineering:

We engineered new features to enhance the model's predictive power. A key feature was the creation of a 7-day rolling average of electricity prices to capture short-term trends.

Time-based features were extracted from the timestamps, including day of the week and month, as these were expected to influence price patterns.

4. Model Selection:

The model selection process involved evaluating different types of models. We decided to use an LSTM (Long Short-Term Memory) neural network due to its effectiveness in capturing sequential patterns in time series data.

LSTM networks were implemented using Python libraries like TensorFlow and Keras.

5.Evaluation Methods:

The dataset was split into training, validation, and testing sets, ensuring that earlier data was used for training and later data for validation and testing.We selected RMSE (Root Mean Squared Error) as the primary evaluation metric to assess how well the model's predictions aligned with actual electricity prices.Backtesting was conducted, involving training the model on a rolling window of historical data and comparing its predictions to actual prices for various time periods.

TUNING:

Hyperparameter tuning was performed to optimize the model. This included adjusting learning rates, the number of LSTM layers, and batch sizes to improve predictive accuracy.

These steps collectively led to the development of a robust electricity price prediction model using AI and Python, enabling us to forecast electricity prices with improved accuracy. The model was then deployed to make real-time predictions and was continuously monitored for updates and refinements.

Creating a detailed Python algorithm for an electricity price prediction project using AI involves several steps.

```
# Import necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from statsmodels.tsa.arima_model import ARIMA

from sklearn.metrics import mean_absolute_error, mean_squared_error

from sklearn.model_selection import train_test_split

from datetime import datetime

from sklearn.preprocessing import MinMaxScaler
```

```
# Load your electricity price dataset (assuming it has columns 'timestamp' and 'price')
data = pd.read_csv('electricity_price_data.csv')
data['timestamp'] = pd.to_datetime(data['timestamp'])
data.set_index('timestamp', inplace=True)

# Data Preprocessing
# Handle missing values
data = data.dropna()

# Feature Engineering (e.g., adding day of the week, time of day)
data['day_of_week'] = data.index.dayofweek
data['hour_of_day'] = data.index.hour

# Normalize the price data
scaler = MinMaxScaler()
data['price'] = scaler.fit_transform(data['price'].values.reshape(-1, 1))

# Split the data into training and testing sets
train_size = int(len(data) * 0.8)
train_data, test_data = data[:train_size], data[train_size:]

# ARIMA Model
model = ARIMA(train_data['price'], order=(5, 1, 0))
model_fit = model.fit(disp=0)

# Make predictions on the test set
predictions, _, _ = model_fit.forecast(steps=len(test_data))
```

```
# Inverse transform predictions to get actual price values
predictions = scaler.inverse_transform(predictions.reshape(-1, 1))

# Evaluate the model
mae = mean_absolute_error(test_data['price'], predictions)
rmse = np.sqrt(mean_squared_error(test_data['price'], predictions))

print(f'Mean Absolute Error: {mae}')
print(f'Root Mean Squared Error: {rmse}')

# Plot the results
plt.figure(figsize=(12, 6))
plt.plot(test_data.index, test_data['price'], label='Actual')
plt.plot(test_data.index, predictions, color='red', label='Predicted')
plt.title('Electricity Price Prediction')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.show()
```

CSV file containing electricity price data with a 'timestamp' and 'price' column. The code performs data preprocessing, splitting, ARIMA modeling, evaluation, and visualization.

The following columns from the electricity price dataset are used:

1. timestamp: This column represents the date and time when the electricity price data was recorded. It is essential for time-series analysis as it helps in understanding how prices vary over time.

2. price: This column contains the actual electricity price values. The program uses this column to train the ARIMA model and make predictions.

Additionally, the program performs some feature engineering:

3. day_of_week: This newly created column represents the day of the week for each timestamp. It's derived from the 'timestamp' column and can be used to capture any day-of-the-week patterns in electricity prices.

4. hour_of_day: This newly created column represents the hour of the day for each timestamp. It can help capture any time-of-day patterns in electricity prices.

The `scaler` is used to normalize the 'price' column to a range between 0 and 1, which is a common practice when working with neural networks and models sensitive to the scale of input data.

These columns and transformations are used to preprocess the data and build an ARIMA model for electricity price prediction. The model is then evaluated, and the results are visualized to assess its performance.

list of the libraries used and instructions on how to install them using the Python package manager, pip:

numpy: NumPy is used for numerical computations, and it provides support for arrays and mathematical functions.

To install NumPy:

```
pip install numpy
```

pandas: Pandas is used for data manipulation and analysis. It provides data structures for efficiently working with structured data, such as data frames.

To install Pandas:

```
pip install pandas
```

matplotlib: Matplotlib is a popular data visualization library, and it's used for creating plots and charts to visualize the data and model results.

To install Matplotlib:

```
pip install matplotlib
```

statsmodels: Statsmodels is used for statistical modeling, including the ARIMA model. It's an essential library for time series analysis.

To install Statsmodels:

```
pip install statsmodels
```

sklearn: Scikit-learn is used for machine learning tasks and metrics. In this example, it's used for metrics like Mean Absolute Error and Mean Squared Error.

To install scikit-learn:

```
pip install scikit-learn
```

datetime: The datetime module is part of Python's standard library and is used for handling date and time-related operations.

No need to install; it's part of the standard library.

MinMaxScaler from sklearn.preprocessing: This is used for normalizing the 'price' column.

Already included if you've installed scikit-learn as mentioned earlier.

To install any of these libraries, open your command prompt or terminal and run the respective pip install command. Make sure you have Python installed on your system.

TRAINING AND TESTING:

the data is split into training and testing sets using the following lines of code:

```
# Split the data into training and testing sets

train_size = int(len(data) * 0.8)

train_data, test_data = data[:train_size], data[train_size:]
```

WORKING:

len(data) calculates the total number of data points in your dataset.

0.8 indicates that 80% of the data will be used for training, and 20% will be used for testing. You can adjust this split ratio to suit your needs.

The train_data variable contains the first 80% of the data, and the test_data variable contains the remaining 20%. This split is a common practice in machine learning to assess the model's performance on unseen data.

The training data (train_data) is used to train the ARIMA model, and the testing data (test_data) is used to evaluate the model's performance by making predictions and comparing them to the actual values.

Hyperparameter tuning for an ARIMA model involves adjusting the model's hyperparameters to find the best configuration for your specific dataset. The key hyperparameters for an ARIMA model are:

1. p (order of Autoregressive component): This parameter represents the number of lag observations in the model. It determines how many past time steps are considered in predicting the current value. A higher `p` means more historical data is used for prediction.

2. d (degree of differencing): This parameter determines the number of times the data is differenced to make it stationary. Stationarity is a key requirement for time series modeling. A higher `d` means more differencing is applied.

3. q (order of Moving Average component): This parameter represents the number of lagged forecast errors used in the prediction. It influences how the model reacts to past forecast errors. A higher `q` incorporates more historical errors.

To tune the hyperparameters:

1. Manual Tuning: You can manually try different combinations of `p`, `d`, and `q`. Start with small values and gradually increase them while evaluating the model's performance on a validation dataset or using cross-validation.

2. Grid Search: You can perform a systematic grid search where you define a range of values for `p`, `d`, and `q`. The grid search will test all possible combinations within these ranges and choose the best performing set of hyperparameters based on a chosen evaluation metric (e.g., RMSE).

3. Automated Tools: Libraries like `pmdarima` in Python offer automated hyperparameter tuning for ARIMA models. They perform a comprehensive search for the best hyperparameters by considering different combinations and picking the best one based on model performance.

Hyperparameter tuning is crucial because the choice of hyperparameters can significantly impact the accuracy of your ARIMA model. The goal is to find the hyperparameters that result in the best predictions for your specific electricity price dataset.

In the provided Python program for electricity price prediction using ARIMA, two common metrics have been used for accuracy checking:

1. Mean Absolute Error (MAE):

- MAE is calculated as the average of the absolute differences between the predicted and actual values. It measures the average magnitude of errors in the predictions. A lower MAE indicates better model accuracy.

In the code:

mae = mean_absolute_error(test_data['price'], predictions)

2. Root Mean Squared Error (RMSE):

RMSE is calculated as the square root of the average of the squared differences between predicted and actual values. It penalizes larger errors more heavily than MAE and is sensitive to outliers.

In the code:

rmse = np.sqrt(mean_squared_error(test_data['price'], predictions))

Both MAE and RMSE provide insights into the accuracy of the model's predictions. A lower value for either of these metrics indicates that the model's predictions are closer to the actual values. These metrics are commonly used for evaluating regression models like the ARIMA model in time series forecasting.

Conclusion:

In summary, the Python program provided offers a foundational framework for an electricity price prediction project using an ARIMA model. The project begins with data preparation, encompassing cleaning and feature engineering, making the dataset conducive for time series analysis. A crucial step is the division of data into training and testing subsets, enabling the ARIMA model to learn from historical patterns and subsequently assess its performance on unseen data. Key evaluation metrics, Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE), gauge the model's predictive accuracy against the actual test data. While the code doesn't explicitly detail hyperparameter tuning, it is an essential process to fine-tune the ARIMA model for optimal performance. Visualizing the results aids in interpretation and evaluation. However, for more complex real-world projects, considerations like seasonality and exogenous variables may come into play, necessitating advanced forecasting techniques. The accuracy of the model hinges on meticulous hyperparameter tuning and comprehensive evaluation, making these steps imperative for constructing a reliable prediction model.