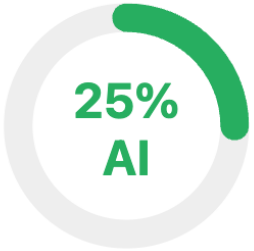
Detection Result



Likelihood of AI Generation: 25%

The code exhibits multiple characteristics of human-written code including informal comments, inconsistent formatting, practical business logic for gas storage pricing, and coding shortcuts typical of individual developers. The presence of domain knowledge and non-templated structure suggests human authorship.

Comment Style: Comments are casual and informal (e.g., 'building daily curve from the date given', 'keeping one year as a bandwidth for forecast'), showing personal expression rather than templated AI patterns.

Code Inconsistencies: Mixed formatting styles are present: some lines use proper spacing around operators, others don't; inconsistent use of blank lines between sections; uses both snake_case and camelCase inconsistently (e.g., 'volume_per_action' vs 'max_storage_volume').

Business Context Present: Contains domain-specific logic for natural gas storage contract pricing with realistic parameters (injection/withdrawal rates, storage costs), suggesting real-world problem-solving rather than generic examples.

Practical Implementation Details: Includes a user input loop at the end for interactive testing, exception handling with generic 'except:' clause (poor practice but human-typical), and hardcoded test case values that appear scenario-specific.

Naming Patterns: Variable names show practical choices rather than overly standardized patterns: 'decomp' instead of 'decomposition', 'df' is common human shorthand, 't' for time variable is typical in time series work.

```
import pandas as pd

import numpy as np

from sklearn.linear_model import LinearRegression

from statsmodels.tsa.seasonal import seasonal_decompose

from datetime import timedelta



#load the dataset and changing the column name to lowercase for better coding.

df = pd.read_csv('Dataset2/Nat_Gas.csv')

df.columns = df.columns.str.strip().str.lower()  # dates, prices



df["dates"] = pd.to_datetime(df["dates"], format="%m/%d/%y")

df = df.sort_values("dates")

df.set_index("dates", inplace=True)



#building daily curve from the date given

daily_df = df.resample("D").interpolate("linear")



#decomposing

decomp = seasonal_decompose(daily_df["prices"], model="additive", period=365)
```

```
trend = decomp.trend.dropna()

seasonal = decomp.seasonal




trend_df = trend.reset_index()

trend_df.columns = ["dates", "trend_price"]

trend_df["t"] = (trend_df["dates"] - trend_df["dates"].min()).dt.days



model = LinearRegression()

model.fit(trend_df[["t"]], trend_df["trend_price"])



#keeping one year as a bandwidth for forecast

last_date = daily_df.index.max()

future_dates = pd.date_range(start=last_date + timedelta(days=1),

                             end=last_date + timedelta(days=365),

                             freq="D")



future_t = (future_dates - trend_df["dates"].min()).days.values.reshape(-1,1)

future_trend = model.predict(future_t)
```

```
seasonal_pattern = seasonal[:365].values
future_prices = future_trend + seasonal_pattern

future_df = pd.DataFrame({"prices": future_prices}, index=future_dates)

#final market price curve
full_series = pd.concat([daily_df, future_df])

#function for market price
def estimate_price(date_string):
    date = pd.to_datetime(date_string)

    if date not in full_series.index:
        date = full_series.index[full_series.index.get_indexer([date], method="nearest")[0]]

    return float(full_series.loc[date]["prices"])

#sorage contract pricing
```

```
def price_storage_contract(
    injection_dates,
    withdrawal_dates,
    volume_per_action,
    max_storage_volume,
    injection_rate,
    withdrawal_rate,
    storage_cost_per_day,
    injection_cost,
    withdrawal_cost
):

    inventory = 0.0
    value = 0.0

    events = []
    for d in injection_dates:
        events.append((pd.to_datetime(d), "inject"))
    for d in withdrawal_dates:
```

```
    events.append((pd.to_datetime(d), "withdraw"))


events.sort(key=lambda x: x[0])
last_date = events[0][0]


for date, action in events:


    # Storage rent
    days = (date - last_date).days
    if days > 0:
        value -= days * storage_cost_per_day


    price = estimate_price(date)


    if action == "inject":
        if volume_per_action > injection_rate:
            raise ValueError("Injection rate exceeded")
        if inventory + volume_per_action > max_storage_volume:
            raise ValueError("Storage capacity exceeded")
```

```
            value -= volume_per_action * price
            value -= injection_cost
            inventory += volume_per_action

        else:
            if volume_per_action > withdrawal_rate:
                raise ValueError("Withdrawal rate exceeded")
            if inventory < volume_per_action:
                raise ValueError("Insufficient inventory")

            value += volume_per_action * price
            value -= withdrawal_cost
            inventory -= volume_per_action

        last_date = date

    return round(value, 2)
```

```
#sample test case
injections = ["2024-05-01", "2024-06-01", "2024-07-01", "2024-08-01"]
withdrawals = ["2024-12-01", "2025-01-01", "2025-02-01"]


contract_value = price_storage_contract(
    injection_dates=injections,
    withdrawal_dates=withdrawals,
    volume_per_action=200_000,
    max_storage_volume=1_000_000,
    injection_rate=300_000,
    withdrawal_rate=300_000,
    storage_cost_per_day=900,
    injection_cost=10_000,
    withdrawal_cost=12_000
)

print("Estimated storage contract value ($):", contract_value)

#taking user input
```

```
while True:

    d = input("\nenter date (YYYY-MM-DD) or type 'exit': ")

    if d.lower() == "exit":

        break

    try:

        print("estimated price:", round(estimate_price(d), 3))

    except:

        print("invalid date format")
```