Project Program Book

Name of the student: Kota MadhaviLatha

Name of the college: K L University

Department: MCA

Specialization: ARTIFICAL INTELLIGENCE

1. Predictive Analytics with Time Series Forecasting

Create a model to forecast future values of a time series, such as stock prices or weather data:

Building a Time Series Forecasting Model:

Time series forecasting involves predicting future values based on historical data. It's widely used in finance, economics, meteorology, and many other fields.

Data Preparation

Before modeling, we need to:

- Import necessary libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal
import  seasonal_decompose
from statsmodels.tsa.stattools
 import adfuller
from sklearn.preprocessing
 import  MinMaxScaler
```

- Load and explore the data:

```
data = pd.read_csv('your_data.csv', index_col='date', parse_dates=True)
data.plot(figsize=(10, 6))
plt.show()
```

Handle missing values:

```
data = data.fillna(method='ffill')  # Or other suitable methods
```

Decompose the time series:

```
decomposition = seasonal_decompose(data, model='additive', period=12)
# Adjust period as needed
```

```
decomposition.plot()
plt.show()
```

- Stationarity check:

```
result = adfuller(data)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
   print('\t%s: %.3f' % (key, value))
```

If the p-value is less than 0.05, the series is likely stationary. Otherwise, differencing or other transformations might be needed.

Model Selection:

Several models can be used for time series forecasting:

Statistical Models

- ARIMA (AutoRegressive Integrated Moving Average): Suitable for stationary time series.
- SARIMA (Seasonal ARIMA): Handles seasonal patterns.
- Exponential Smoothing: Good for short-term forecasting.

Machine Learning Models

- Support Vector Regression (SVR): For non-linear relationships.
- Random Forest: Handles complex patterns.
- Neural Networks (LSTM, GRU): Capture long-term dependencies.

Model Training and Evaluation

- Split data into training and testing sets:

```
from sklearn.model_selection
 import train_test_split X_train, X_test, y_train, y_test = train_test_split(X,   y, test_size=0.2, shuffle=False)
```

- Model fitting and evaluation:

```
from sklearn.metrics
 import mean_squared_error
# Fit the model
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)
# Evaluate
mse = mean_squared_error(y_test, y_pred)
print('Mean Squared Error:', mse)
```

Example using ARIMA:

```
from statsmodels.tsa.arima_model
import ARIMA
# Assuming data is stationary
model = ARIMA(data, order=(p, d, q))  # Replace p, d, q with optimal values
model_fit = model.fit(disp=0)
forecast = model_fit.forecast(steps=12)
# Forecast for the next 12 periods
```

Visualization:

- Plot actual vs. predicted values.
- Calculate forecasting error metrics (MAE, RMSE, etc.).

Additional Considerations:

- Feature Engineering: Incorporate external factors (e.g., economic indicators, weather data).
- Hyperparameter Tuning: Optimize model parameters for better performance.
- Model Selection: Experiment with different models to find the best fit.
- Ensemble Methods: Combine multiple models for improved accuracy.

Data Collection: Obtain a time series dataset (e.g., historical stock prices from Yahoo Finance):

Obtaining a Time Series Dataset: A Case Study with Yahoo Finance

Before diving into the code, let's clarify what kind of data we're looking for:

- Stock symbol: The unique identifier for the company (e.g., AAPL for Apple).
- Start date: The beginning of the desired time period.
- End date: The end of the desired time period.
- Data points: The specific information we want (e.g., Open, High, Low, Close, Adjusted Close, Volume).

Python Library: yfinance

For this task, we'll use the yfinance library, which provides a convenient interface to Yahoo Finance's data.

```
import yfinance as yf

import pandas as pd

# Define the stock symbol, start date, and end date

ticker = "AAPL"

start_date = "2018-01-01"

end_date = "2023-12-31"

# Download the data

data = yf.download(ticker, start=start_date, end=end_date)

# Display the first few rows

print(data.head())
```

Data Exploration and Cleaning:

Once you have the data, it's essential to explore it:

- Check for missing values:

```
print(data.isnull().sum())
```

- Handle missing values:

data = data.fillna(method='ffill') # Or other suitable methods

- Visualize the data:

```
import matplotlib.pyplot as plt

data['Close'].plot(figsize=(10, 6))

plt.title(f"{ticker} Closing Price")

plt.show()
```

Additional Considerations:

- Data Frequency: You can specify the frequency of the data using the interval parameter in yf.download. For example, interval="1m" for 1-minute data, interval="1h" for 1-hour data.
- Multiple Stocks: You can download data for multiple stocks at once by passing a list of tickers to yf.download.
- Data Storage: Consider saving the data to a CSV or other format for future use.
- Data Enrichment: You might want to add additional features, such as technical indicators or fundamental data.

Data Preprocessing: Handle missing values, perform normalization, and create features like moving averages.

1. Data Path: The location of your time series data (e.g., CSV file, Excel file, database).
2. Data Format: The structure of your data (e.g., column headers, date format).
3. Target Variable: The specific column you want to forecast (e.g., 'Close' price for stock data).

Once I have this information, I can demonstrate the following preprocessing steps:

Handling Missing Values:

- Identifying missing values: We'll use pandas to check for null values in the dataset.

- Imputation: Techniques like filling missing values with the previous value (forward fill), next value (backward fill), or mean/median imputation can be applied based on the data characteristics.

Normalization:

- Scaling: We'll use techniques like Min-Max scaling or Standardization to bring data within a specific range or distribution. This is crucial for many machine learning algorithms.

Creating Features:

- Moving averages: We'll calculate simple moving averages (SMA) and exponential moving averages (EMA) for different window sizes to capture trends in the data.
- Other features: Depending on the data and problem, we might consider creating additional features like differences, returns, or lagged values.

Example Code (Assuming data is in a CSV file):

```
import pandas as pd

import numpy as np

# Replace 'your_data.csv' with the actual path

data = pd.read_csv('your_data.csv', index_col='date', parse_dates=True)

# Handle missing values (example: forward fill)

data = data.fillna(method='ffill')

# Normalize data (example: Min-Max scaling)

from sklearn.preprocessing

 import MinMaxScaler

scaler = MinMaxScaler()

data_scaled = scaler.fit_transform(data)

# Create moving averages

data['SMA_20'] = data['Close'].rolling(window=20).mean()

data['EMA_12'] = data['Close'].ewm(span=12).mean()
```

```
from sklearn.metrics import precision_score, recall_score, f1_score

# Assuming y_true is the ground truth and y_pred is the model's predictions
y_true = y_test
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, zero_division=1)
recall = recall_score(y_true, y_pred, zero_division=1)
f1 = f1_score(y_true, y_pred, zero_division=1)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

```
Accuracy: 0.0
Precision: 0.0
Recall: 1.0
F1-score: 0.0
```

Model Building: Choose a model like ARIMA, LSTM (Long Short-Term Memory), or Prophet for forecasting:

The optimal model for your time series data depends on several factors:

- Stationarity: Is the data stationary (mean and variance constant over time)?
- Seasonality: Does the data exhibit seasonal patterns?
- Trend: Is there an upward or downward trend in the data?
- Complexity: How complex are the patterns in the data?
- Data Size: How much data is available?

Model Options:

ARIMA (AutoRegressive Integrated Moving Average)

- Best for: Stationary time series with linear relationships.

1. Introduction to ARIMA: nonseasonal models - Duke People

Pros: Simple to understand and implement, often effective for short-term forecasting.

- Cons: Assumes linearity, might struggle with complex patterns and non-stationarity.

LSTM (Long Short-Term Memory)

- Best for: Non-linear, complex patterns, and long-term dependencies.
- Pros: Can capture complex relationships, effective for long-term forecasting.

1. Long short-term memory :

- Cons: Requires more computational resources, hyperparameter tuning can be challenging.

Prophet:

- Best for: Time series with strong seasonal components and trends.
- Pros: Easy to use, handles missing data well, incorporates external factors.
- Cons: Might be less accurate for highly complex patterns.

Model Selection Process:

1. Exploratory Data Analysis (EDA): Understand the characteristics of your data.
2. Stationarity Check: Determine if the data is stationary. If not, apply differencing or other transformations.
3. Model Selection: Choose a model based on the characteristics of your data.
4. Hyperparameter Tuning: Optimize model parameters for best performance.
5. Evaluation: Compare models using metrics like RMSE, MAE, or MAPE.

Example Code Structure:

```
import pandas as pd
import numpy as np
from statsmodels.tsa.arima_model
import ARIMA
from sklearn.preprocessing
 import MinMaxScaler
from tensorflow.keras.models
import Sequential
from tensorflow.keras.layers
import LSTM, Dense
from fbprophet import Prophet
# Load your preprocessed data here
# ARIMA Model
model_arima = ARIMA(data, order=(p, d, q))
 # Replace with optimal parameters
model_fit = model_arima.fit()
forecast_arima = model_fit.forecast(steps=forecast_horizon)
# LSTM Model
# Reshape data for LSTM
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)
# Create LSTM model
model_lstm = Sequential([
   LSTM(units=50, return_sequences=True, input_shape=(look_back, 1)),
   LSTM(units=50),
   Dense(1)
])
```

```
# Compile and fit the model

# Prophet Model

df = pd.DataFrame({'ds': data.index, 'y': data['target_column']})

model_prophet = Prophet()

model_prophet.fit(df)

future = model_prophet.make_future_dataframe(periods=forecast_horizon)

forecast_prophet = model_prophet.predict(future)
```

Training: Train the model on the historical data and validate its performance using a hold-out validation set:

Training and Validation:

Data Splitting:

The first step is to divide the dataset into training and validation sets. Typically, a 70/30 or 80/20 split is used.

```
import pandas as pd

from sklearn.model_selection

import train_test_split

# Assuming your data is in a pandas DataFrame

data = pd.read_csv('your_data.csv', index_col='date', parse_dates=True)

# Split into features (X) and target variable (y)

X = data.drop('target_column', axis=1)

 # Replace 'target_column' with your target

y = data['target_column']


# Split into training and validation sets

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,
shuffle=False)
```

Model Training:

Next, we train the chosen model on the training set.

ARIMA Example:

```
from statsmodels.tsa.arima_model
 import ARIMA
model = ARIMA(y_train, order=(p, d, q))
 # Replace with optimal parameters
model_fit = model.fit()
```

LSTM Example:

```
from tensorflow.keras.models
 import Sequential
from tensorflow.keras.layers
import LSTM, Dense
# Assuming data is preprocessed and reshaped for LSTM
model = Sequential([LSTM(units=50, return_sequences=True,
input_shape=(look_back, 1)),
   LSTM(units=50),
   Dense(1)
])
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

Prophet Example:

```
from fbprophet
 import Prophet
df_train = pd.DataFrame({'ds': X_train.index, 'y': y_train})
model = Prophet()
model.fit(df_train)
```

Model Validation:

Evaluate the model's performance on the validation set.

```
# Make predictions on the validation set
y_pred = model.predict(X_val)
# For ARIMA and LSTM
# For Prophet, use model.predict(df_val) where df_val has 'ds' column
from sklearn.metrics import mean_squared_error, mean_absolute_error
mse = mean_squared_error(y_val, y_pred)
mae = mean_absolute_error(y_val, y_pred)
print('MSE:', mse)
print('MAE:', mae)
```

Important Considerations:

- Hyperparameter Tuning: Experiment with different model parameters to improve performance.
- Evaluation Metrics: Choose appropriate metrics based on the problem (e.g., RMSE, MAE, MAPE).
- Overfitting: Watch for signs of overfitting (e.g., low training error, high validation error).
- Feature Engineering: Create additional features to improve model performance.

Forecasting: Use the model to predict future values and visualize the forecast against the actual data.

Forecasting and Visualization:

Generating Forecasts:

Once you have a trained model, the next step is to generate predictions for future time periods.

ARIMA

```
from statsmodels.tsa.arima_model
 import ARIMA
# Assuming model_fit is your trained ARIMA model
forecast, stderr, conf_int = model_fit.forecast(steps=forecast_horizon)
```

LSTM:

```
from numpy import array

# Assuming your data is prepared in the correct format

X_test = array(X_test)

X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

forecast = model.predict(X_test)
```

Prophet:

```
from fbprophet

 import Prophet

future = model.make_future_dataframe(periods=forecast_horizon)

forecast = model.predict(future)
```

Visualizing Forecasts:

```
import matplotlib.pyplot as plt

# Assuming y_true is your actual data and forecast is your predicted data

plt.figure(figsize=(12, 6))

plt.plot(y_true, label='Actual')

plt.plot(forecast, label='Forecast')

plt.legend()

plt.show()
```

Additional Considerations:

- Confidence Intervals: Some models, like ARIMA and Prophet, provide confidence intervals. You can plot these to visualize the uncertainty in the forecast.
- Error Metrics: Calculate metrics like Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE) to evaluate the forecast accuracy.
- Dynamic Forecasting: For models like ARIMA, consider updating the model with new data as it becomes available for more accurate forecasts.

- Model Comparison: If you've trained multiple models, compare their forecasts visually and using error metrics to select the best one.

Example with Confidence Intervals:

Assuming you have confidence intervals stored in conf_int

plt.fill_between(forecast.index, conf_int[:, 0], conf_int[:, 1], color='gray', alpha=0.3)

```
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.metrics import Precision, Recall, BinaryAccuracy

# Example DataFrame (replace this with the actual DataFrame 'df')
df = pd.DataFrame({
    'text': ["I love this movie", "I hate this movie", "This movie is great", "This movie is terrible"],
    'sentiment': [1, 0, 1, 0]
})

# Preprocess the data
X = df['text']
y = df['sentiment']

# Tokenize the text
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(X)
X = tokenizer.texts_to_sequences(X)
X = pad_sequences(X, padding='post')

# Define the test size
test_size = 0.2
```

```python
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42)

# Build the model
model = Sequential()
model.add(Embedding(input_dim=5000, output_dim=64, input_length=X.shape[1]))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[Precision(), Recall(), BinaryAccuracy()])

# Train the model
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))

# Evaluate on test data
test_loss, test_precision, test_recall, test_accuracy = model.evaluate(X_test, y_test)

print("Test Loss:", test_loss)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
print("Test Accuracy:", test_accuracy)
```

```
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step - binary_accuracy: 0.3333 - loss: 0.6940 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.6976 - val_pre
Epoch 2/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 609ms/step - binary_accuracy: 0.6667 - loss: 0.6907 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7037 - val_precisi
Epoch 3/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 144ms/step - binary_accuracy: 0.6667 - loss: 0.6875 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7099 - val_precisi
Epoch 4/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step - binary_accuracy: 0.6667 - loss: 0.6843 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7162 - val_precisi
Epoch 5/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 131ms/step - binary_accuracy: 0.6667 - loss: 0.6811 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7227 - val_precisi
Epoch 6/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 123ms/step - binary_accuracy: 0.6667 - loss: 0.6779 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7295 - val_precisi
Epoch 7/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 104ms/step - binary_accuracy: 0.6667 - loss: 0.6745 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7366 - val_precisi
Epoch 8/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 109ms/step - binary_accuracy: 0.6667 - loss: 0.6711 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7441 - val_precisi
Epoch 9/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 134ms/step - binary_accuracy: 0.6667 - loss: 0.6676 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7521 - val_precisi
Epoch 10/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 140ms/step - binary_accuracy: 0.6667 - loss: 0.6639 - precision: 0.6667 - recall: 1.0000 - val_binary_accuracy: 0.0000e+00 - val_loss: 0.7605 - val_precisi
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - binary_accuracy: 0.0000e+00 - loss: 0.7605 - precision: 0.0000e+00 - recall: 0.0000e+00
Test Loss: 0.7604998350143433
Test Precision: 0.0
Test Recall: 0.0
Test Accuracy: 0.0
```