ICT607: Artificial Intelligence for Cybersecurity

Experiment 10

1 AI-driven anomaly detection: leveraging ARIMA forecasting technique

Forecasting is a technique used to predict future trends based on historical data. In the context of cybersecurity, forecasting can be used to identify patterns and anomalies that may indicate a potential security threat. By analyzing patterns in data over time, forecasting algorithms can help detect anomalies in real-time, allowing organizations to proactively respond to potential security incidents before they become major issues.

In this lab, we will use ARIMA (Autoregressive Integrated Moving Average) forecasting algorithm. ARIMA is a commonly used time series forecasting model. It is a powerful statistical technique for analysing and predicting patterns in data, particularly in cases where the data is non-stationary or has a trend.

ARIMA models are based on three components:

- Autoregression (AR): A linear regression model that uses past values to predict future values. An AR(p) model uses p past values to predict the next value in the series. For example, an AR(2) model for the stock price of a company would use the last two days' stock prices to predict the next day's stock price.
- Integrated (I): Integration involves transforming the time series to make it stationary, which means that its statistical properties (such as mean and variance) do not change over time. This is important because many time series have trends or seasonal patterns that make them non-stationary. One common way to transform a time series is to difference it, which involves subtracting each observation from the previous observation. If the original time series is denoted by Y_t, the differenced time series is denoted by $Y_t - Y_t - 1$. For example, if you were trying to forecast the number of monthly visitors to a website, differencing the data could help remove any trends or seasonality.
- Moving Average (MA): A model that uses past forecast errors to predict future values. In an MA(q) model, the next value in the series is a function of the past q errors. For example, an MA(1) model for the daily temperature would use yesterday's forecast error to predict today's temperature.

ARIMA models use a combination of these three components to model the time series data and predict future values.

Reference book: Chio, C., & Freeman, D. (2018). *Machine learning and security: Protecting systems with data and algorithms* (First edition). O'Reilly Media. (page 95)

Reference programs: https://github.com/oreilly-mlsec/book-resources/blob/master/chapter3/arima-forecasting.ipynb (Modified and bug fixed)

[ ]:
```
# install pyflux (required for using ARIMA model) at colab
!pip install git+https://github.com/RJT1990/pyflux
```

[2]:
```
import pandas as pd
from datetime import datetime
import pyflux as pf
import matplotlib.pyplot as plt
```

1.1 Dataset

Download the datasets and keep it on your Google drive folder (e.g., `Colab Notebooks/AICS/` folder)

[3]:
```
from google.colab import drive
drive.mount("/content/drive")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[4]:
```
# Read in the training and testing dataset files
data_train_a = pd.read_csv(
      '/content/drive/MyDrive/Colab Notebooks/AICS/cpu-train-a.csv',
      parse_dates=[0],
      infer_datetime_format=True
  )

data_test_a = pd.read_csv(
      '/content/drive/MyDrive/Colab Notebooks/AICS/cpu-test-a.csv',
      parse_dates=[0],
      infer_datetime_format=True
  )
```

The second and third parameters `parse_dates` and `infer_datetime_format` are used to inform pandas how to handle the date/time information in the CSV file.

The `parse_dates` parameter in this case indicates that pandas should parse the first column of the CSV file as dates.

The `infer_datetime_format` parameter is True, which means that pandas will attempt to infer the date/time format from the CSV file.

[5]:
```
data_train_a.sample(5)
```

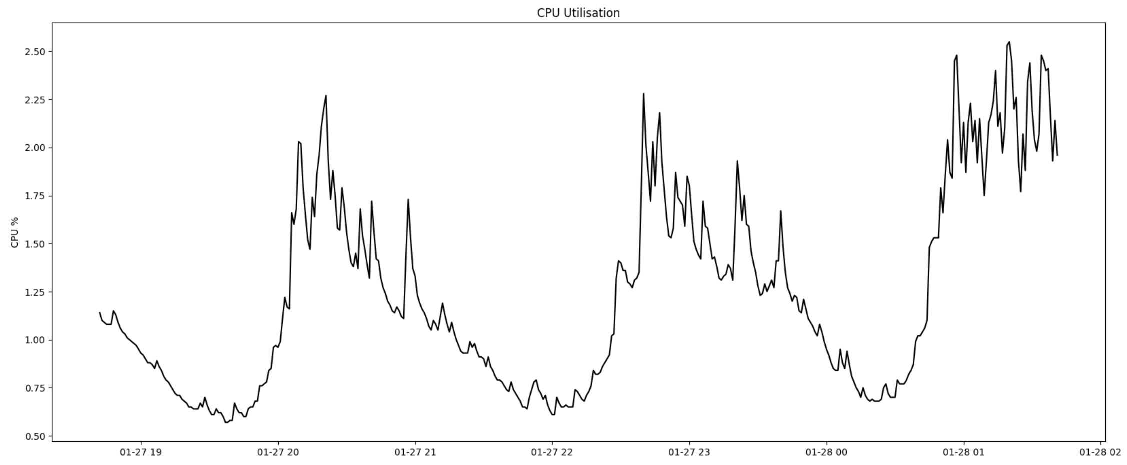
[5]:
```
               datetime   cpu
217 2017-01-27 22:19:00  0.82
33  2017-01-27 19:15:00  0.72
273 2017-01-27 23:15:00  1.33
302 2017-01-27 23:44:00  1.24
248 2017-01-27 22:50:00  1.64
```

[6]:
```
print('Train data shape:', data_train_a.shape)
print('Test data shape:', data_test_a.shape)
```

```
Train data shape: (420, 2)
Test data shape: (60, 2)
```

[7]:
```
plt.figure(figsize=(20,8))
plt.plot(data_train_a['datetime'], data_train_a['cpu'], color='black')
plt.ylabel('CPU %')
plt.title('CPU Utilisation')
```

[7]:
```
Text(0.5, 1.0, 'CPU Utilisation')
```



1.2 Build the ARIMA model

[8]:
```
model_a = pf.ARIMA(
    data=data_train_a,
    ar=11,
    ma=11,
    integ=0,
    target='cpu'
)
```

Documentation of ARIMA implementation by pyflux is here.

There are some tricks to determining the AR and MA orders and the differencing order. But you can also try with different values to see what work best for the dataset – hyperparameter tuning.

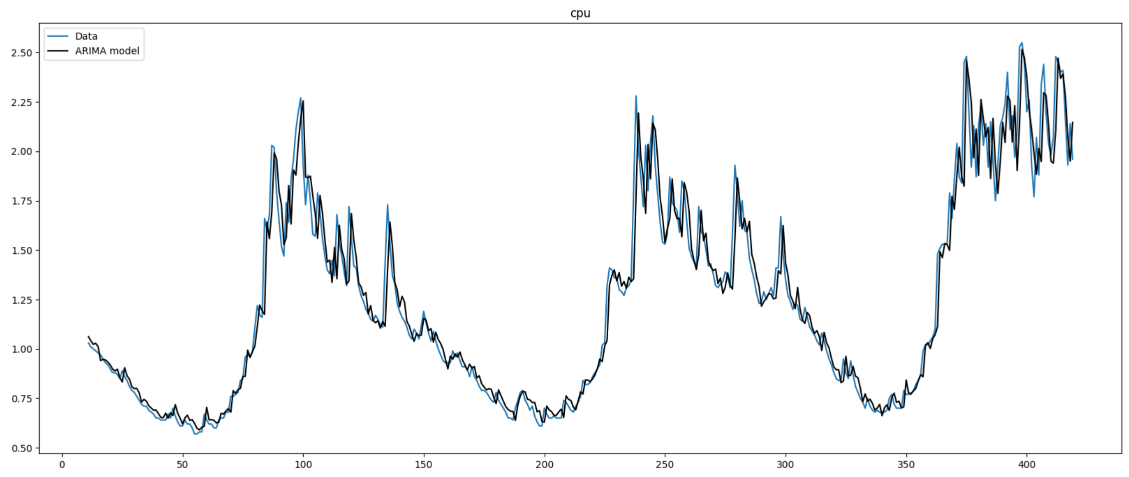
To oversimplify matters, AR and MA orders are needed to correct any residual autocorrelations that remain in the differenced series (i.e., between the time-shifted series and itself). The differencing order is a term used to make the series stationary—an already stationary series should have a differencing order of 0, a series with a constant average trend (steadily trending upward or downward) should have a differencing order of 1, etc.

1.3 Running the ARIMA model with necessary plotting

[9]:
```python
# Estimate latent variables for the model using the
# Metropolis-Hastings algorithm as the inference method
x = model_a.fit("M-H")
```

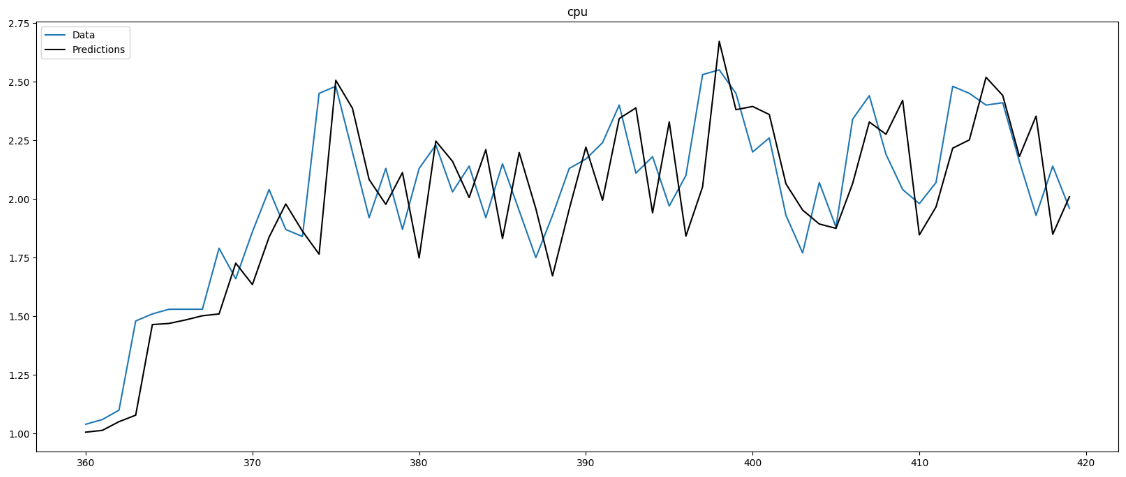
```
Acceptance rate of Metropolis-Hastings is 0.0
Acceptance rate of Metropolis-Hastings is 0.0776
Acceptance rate of Metropolis-Hastings is 0.0949
Acceptance rate of Metropolis-Hastings is 0.1723
Acceptance rate of Metropolis-Hastings is 0.2684

Tuning complete! Now sampling.
Acceptance rate of Metropolis-Hastings is 0.419225
Acceptance rate of Metropolis-Hastings is 0.2217
Acceptance rate of Metropolis-Hastings is 0.27445
```

[10]:
```python
# Plot the fit of the ARIMA model against the data
model_a.plot_fit(figsize=(20,8))
```



As we can see, the results fit the observed data quite well. Next, we can do an in-sample test on the last 60 data points of the training data. The in-sample test is a validation step that treats the last subsection of the series as unknown and performs forecasting for those time steps. This process allows us to evaluate performance of the model without running tests on future/test data:

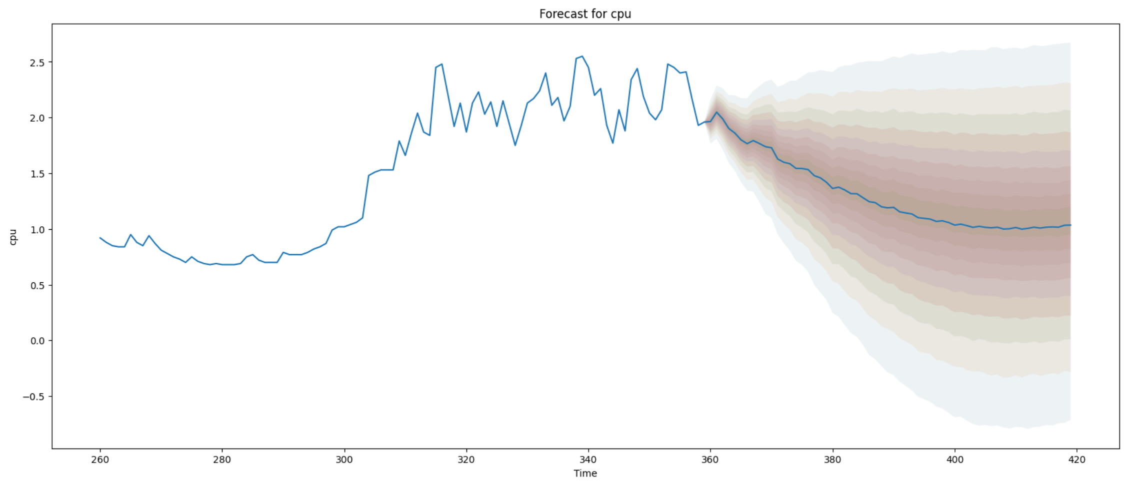
[11]:
```python
model_a.plot_predict_is(h=60, figsize=(20,8))
```



The in-sample prediction test looks pretty good because it does not deviate from the original series significantly in phase and amplitude.

Now, let's run the actual forecasting, plotting the most recent 100 observed data points followed by the model's 60 predicted values along with their confidence intervals.

[12]:
```
model_a.plot_predict(h=60, past_values=100, figsize=(20,8))
```

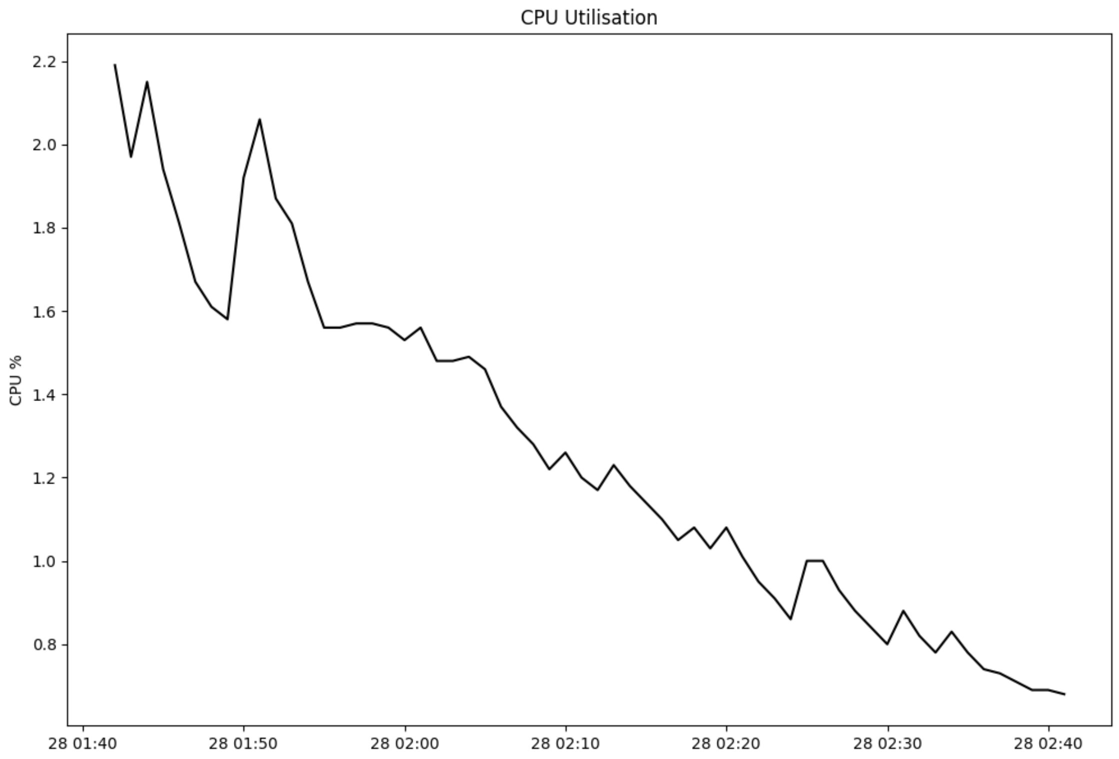


Bands with a darker shade imply a higher confidence.

Now, let's compare the predicted values with actual values to detect anomalies. Actual observed values corresponding to the above 60 predicted values are stored in the `data_test_a` variable.

[13]:
```
plt.figure(figsize=(12,8))
plt.plot(data_test_a['datetime'], data_test_a['cpu'], color='black')
plt.ylabel('CPU %')
plt.title('CPU Utilisation')
```

```
[13]: Text(0.5, 1.0, 'CPU Utilisation')
```



Since the forecasted CPU utilisation ad observed CPU utilisation are quite similar and falls under higher confidence bands, it is unlikely that there are any anomalies.

1.4 Practice task

Use the `cpu-train-b.csv` datasets to fit an ARIMA model and then detect anomalies on `cpu-test-b.csv` dataset.