europython
Edinburgh 23-29 July
2018

Create your own Artificial Intelligence to monitor your Linux System!

Maha Mdini, PhD student, IMT Atlantique,
Data engineer, Astellia/Exfo

ABSTRACT

Manual monitoring of Linux systems consists on monotonous and repetitive tasks that may take some effort and time to the user. In order to automate these processes, one may think of simple artificial intelligence techniques that are easily understood, implemented and tested by the user. This poster proposes some examples of straightforward techniques based on statistical calculations to automate system troubleshooting and performance monitoring. The use cases explored here are just simple examples and more elaborate ideas could be thought of.

Linux

Linux gathers data about the system and the running applications in log files. These files help in troubleshooting the system, understanding security issues, monitoring services and applications and modeling the user behavior. Log files are located in the repository \var\log of the Linux file system. They keep track of important events related to services, applications, the system and the Linux kernel. By analyzing these data, one may gain a clear insight about the system in hands.

Artificial Intelligence

Artificial Intelligence consists on statistical tools and Machine Learning techniques that enable the computer to make "decisions" based on data. In this poster, we are using Python packages that contain a variety of functions to analyze data. The following instructions are used to import the packages needed for our analysis.

```
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as sp
from sklearn.decomposition import PCA
import os, re
```

References

www.eurovps.com/blog/important-linux-log-files-you-must-be-monitoring/

Troubleshooting

/var/log/kern.log
This file contains events logged by the Linux kernel. The logged warnings and errors help to troubleshoot the kernel and debug hardware and connectivity issues.
Errors do not mean the existence of a real issue. For example, we may have logged connectivity errors before a wireless connection is correctly established. In order to detect real problems, we may study the distribution of the number of errors during a period of time (learning phase), then trigger an alert when an error occur more times than expected. To do so, we start by parsing the log file.

```
file= '/var/log/kern.log'
with open(file) as f:
    content = f.readlines()
content = [x.strip() for x in content]
times= [datetime.strptime(i[:15], '%b %d %H:%M:
%S').replace(year=2018) for i in content]
events = [x[65:] for x in content]
df=pd.DataFrame(events,index=times, columns=['events'])
```

Then we count the number of occurrences of each event:

```
S=df['events'].value_counts()
```

Here is an example of what we may get:

```
atkbd serio0: Unknown key released          287
ACPI: Power Resource [WRST] (on)            171
ACPI: Dynamic OEM Table Load:                63
IPv6: wlp1s0: link is not ready              61
```

Then, we save S into a file which name is the current date:

```
S.to_csv('learning/'+str(datetime.now())+'.csv')
```

We run this procedure every time we boot the computer. We may use crontab for that. Then when we have enough data, we can create our model (distribution). First, we start by reading all the data into one data frame L:

```
L=pd.DataFrame()
for file in os.listdir('learning'):
    z=pd.read_csv('learning/'+file, names=[file])
    L=pd.concat([L, z], axis=1)
```

Each row of L contains an event (the index) and its number of occurrences at each date.
For each event, we assume that it has a Gaussian distribution and we estimate its parameters (mean, standard deviation) and store them in a data frame P.

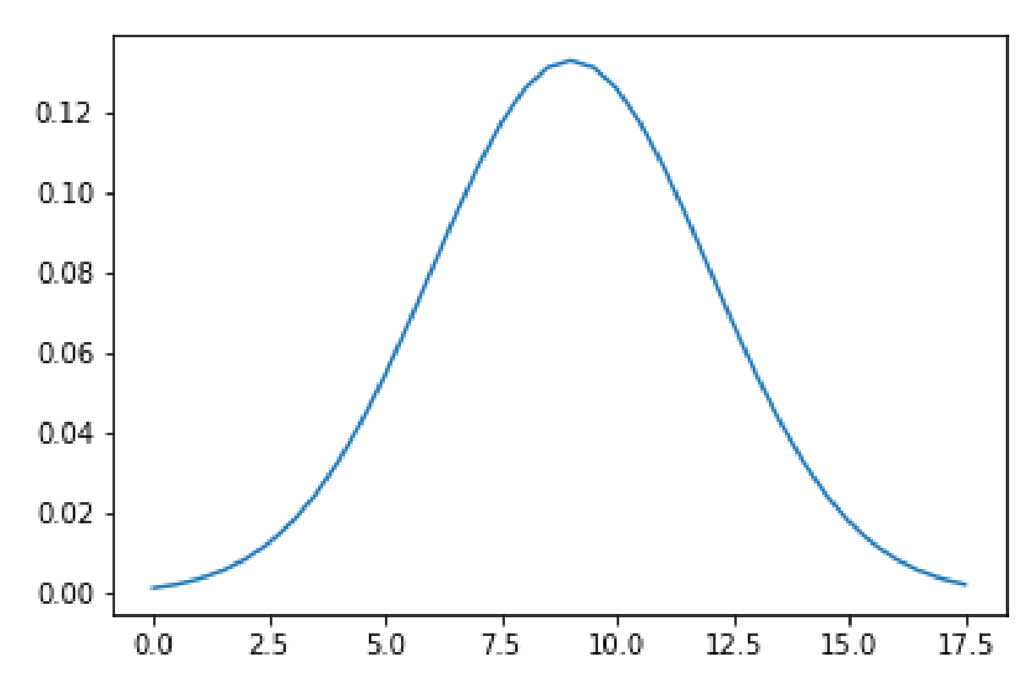
```
P=pd.DataFrame(columns=['mu','std'])
for index, row in L.iterrows():
    mu=np.mean(row.values)
    s=np.std(row.values)
    P.loc[index]=[mu,s]
```

Here is the code to plot the distribution of an example of errors: ''hp_wmi: query 0xd returned error 0x5''

```
mu,s=P.loc['hp_wmi: query 0xd returned error 0x5']
x_axis= np.arange(mu-3*s,mu+3*s,.5)
plt.plot(x_axis,sp.norm.pdf(x_axis,mu,s))
```

Here is the graph of the theoretical distribution:

Figure 1. The Probability Density Function of the number of occurrences of an error



To detect that an anomaly (an error/warning occurring more than expected), we set a threshold equal mu+ 3std.

```
P['threshold']=P['mu']+3*P['std']
```

This is the end of the learning phase. To detect anomalies in real time, we read new data as explained before in the first code block, calculate the number of occurrences of each event S (second code block), add it to P as the real time values:

```
P['real_time']=S
```

We trigger an alert if a real time value is greater than its corresponding threshold.

```
P['alert']= P['real_time']>P['threshold']
```

A customized messaging system could be attached to this algorithm and alerts the user every time an anomaly occurs.
This solution is simple and applicable to any log file containing events repeatedly occurring within the same range of times such as *var/log/boot.log* .

Performance

Sar command line collects performance data in real time to monitor the CPU, memory and I/O.
To create a log file containing CPU performance:

```
sar -u 1 4000 > sar.log
```

Then, we read the file and parse data, to create a data frame:

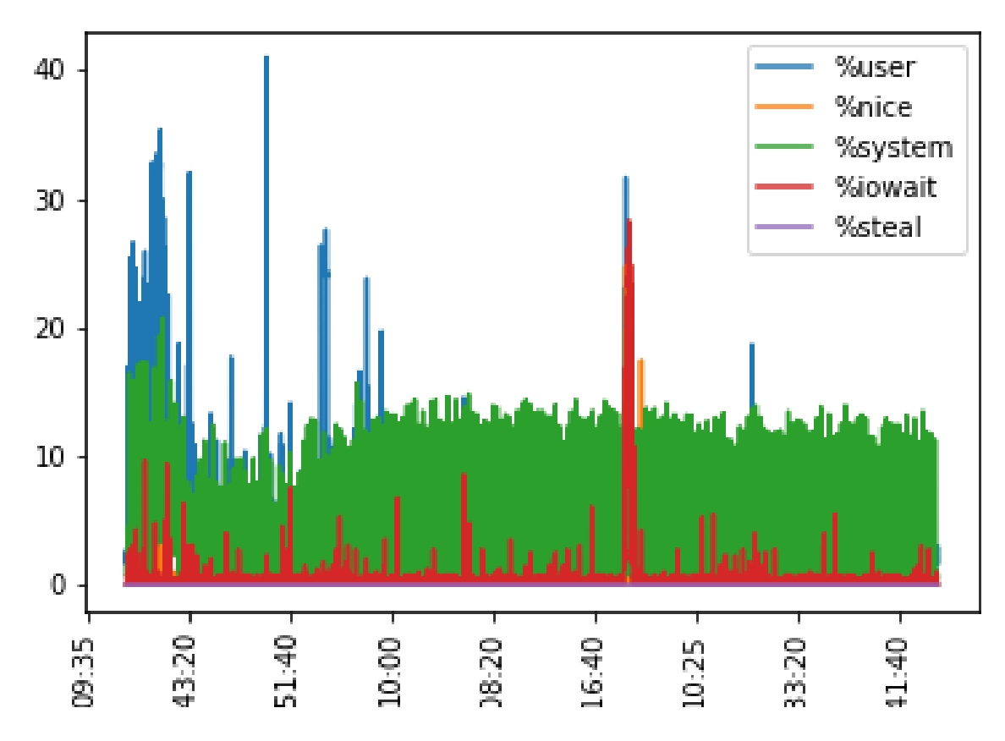
```
file='sar.log'
with open(file) as f:
    content = f.readlines()
content = [x.strip() for x in content]
content = [x.replace(',', '.') for x in content]
content=content[2:-1]
content = [re.split(' *', x) for x in content]
columns= content[0]
df = pd.DataFrame(content[1:], columns=columns)
df.columns.values[0] = 'time'
df['time']=df['time'].apply(lambda x:datetime.strptime(x, '%H:%M:
%S').time())
df = df.set_index('time')
df.drop(['CPU','%idle'],axis=1,inplace=True)
df=df.astype('float')
```

At this level, we have a data frame of CPU usage by the user, the system, I/O waiting processes, nice (process with modified priority) .

```
df.plot()
plt.xticks(rotation=90)
```

Here is the output data frame:

Figure 2. CPU usage by different tasks.

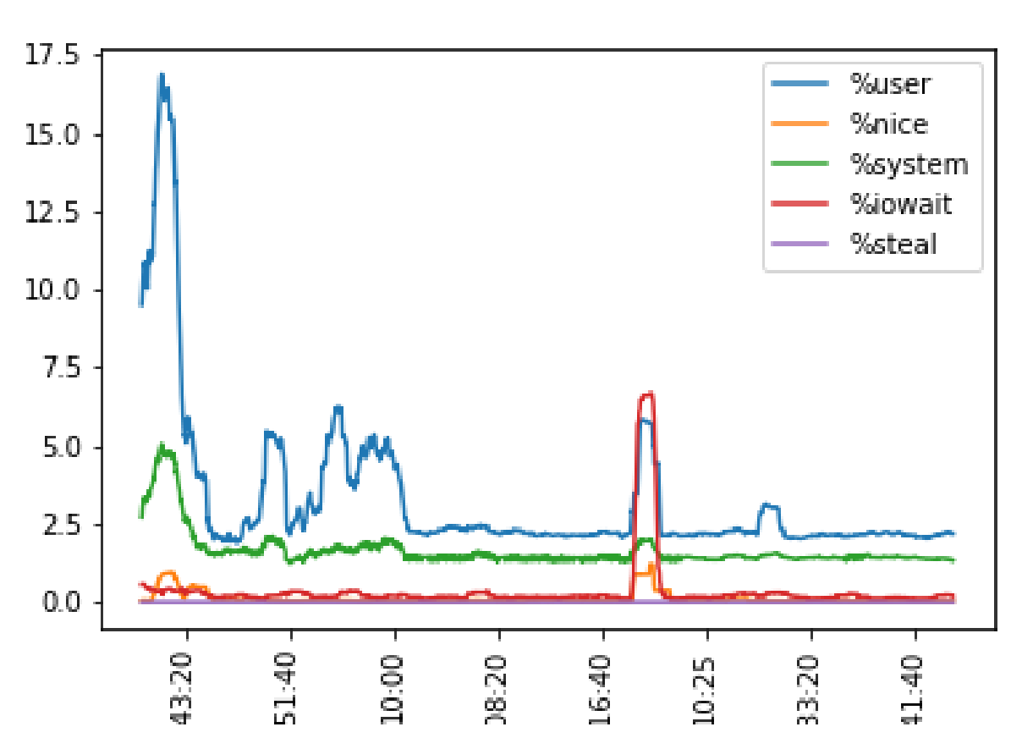


The data contain many peaks. To put the data frame into a more understandable shape, we apply the **moving average**:

```
df=df.rolling(window=100).mean()
df.dropna(inplace=True)
```

Here is the output data frame:

Figure 3. The moving average of the CPU usage.

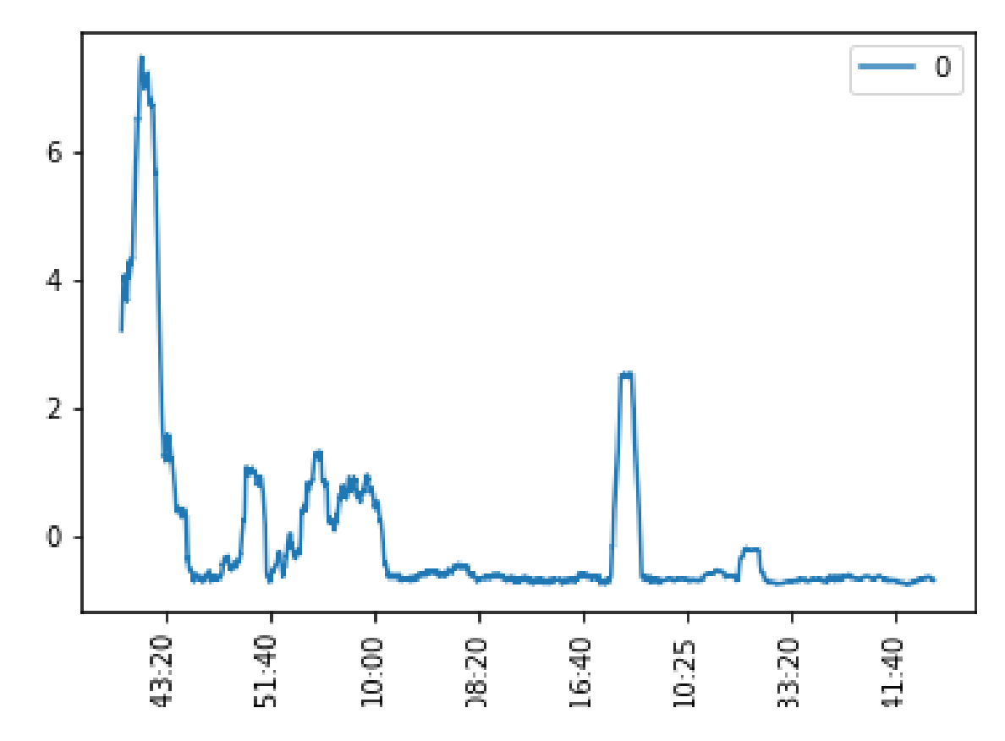


In order to detect peaks of CPU usage, we may apply Principal Component Analysis (PCA), to project all the components on one axis containing the maximum of variance of the 5 components. (The PCA is based on linear combinations) .

```
pca = PCA(n_components=1)
res=pca.fit_transform(df)
```

Here is the result:

Figure 4. PCA projection.



To see the coordinates of the new axis and the variance kept in the PCA projection:

```
print(pca.components_, pca.explained_variance_ratio_)
```

Now, we can set a threshold in this axis to detect CPU usage peaks and then alert the user.

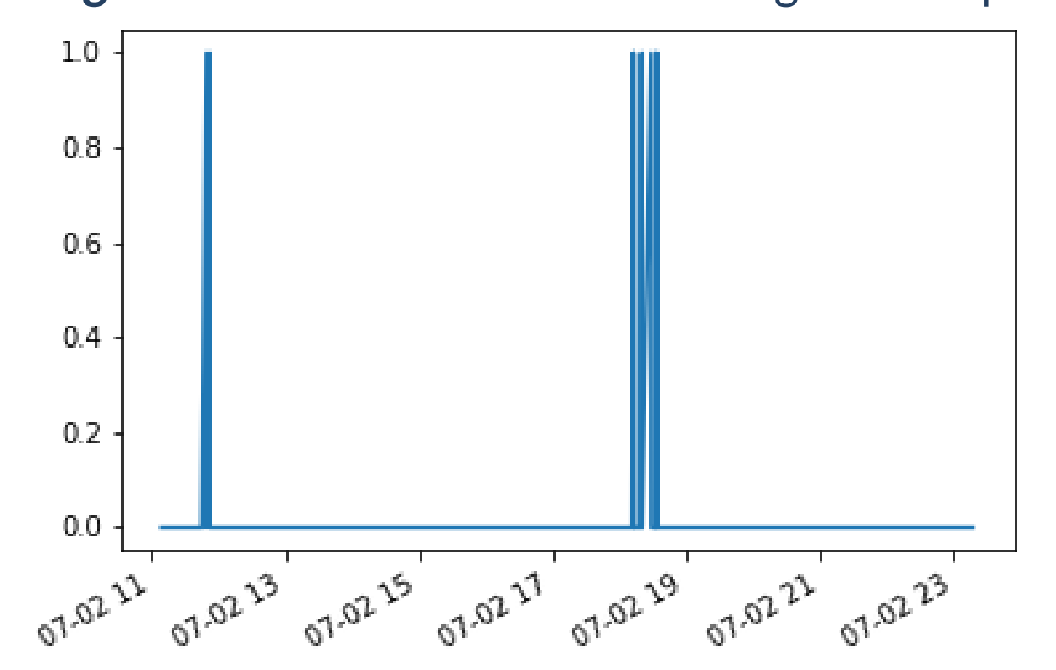
Security

/var/log/auth.log
This file contains authentication attempts and user authorization related events in Debian like systems.
In CentOS systems, this file is named *var/log/secure*.
By tracking login attempts, we can detect attacks related to authentication/authorization such as brute force attacks.
To do so, we start by parsing the log file and extracting a time series containing the number of failed login attempts.

```
file= '/var/log/auth.log'
with open(file) as f:
    content = f.readlines()
content = [x.strip() for x in content]
times= [datetime.strptime(i[:15], '%b %d %H:%M:%S')
        .replace(year=2018) for i in content]
failures=[1 if 'failure' in c else 0 for c in content]
S= pd.Series(failures, index=times)
S.plot()
```

Here is an example of what we may obtain:

Figure 5. A time series of failed login attempts.

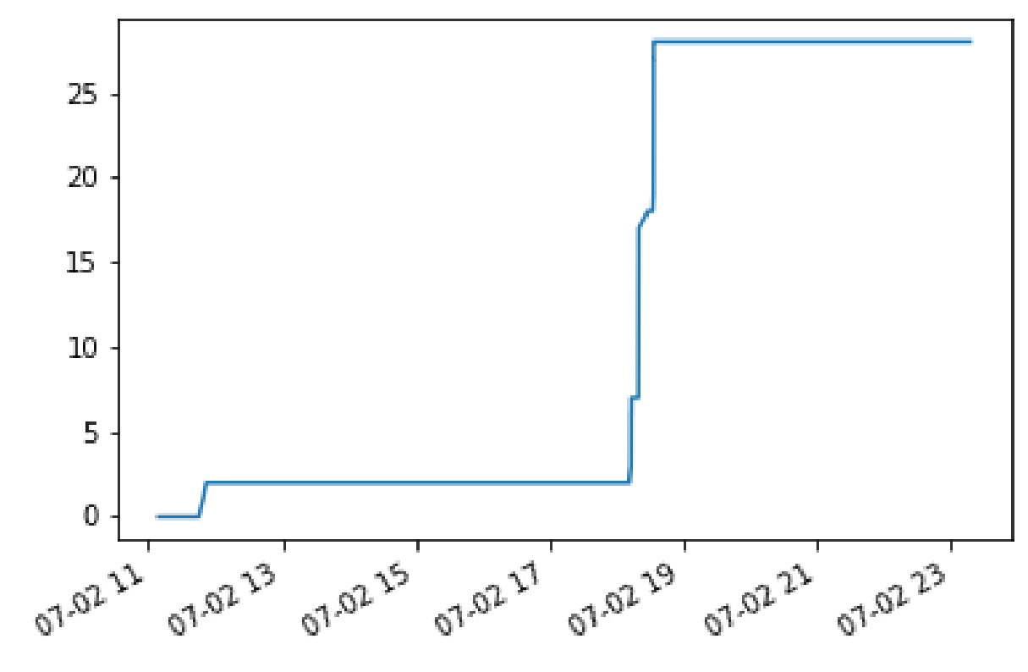


The isolated failed login attempt could be seen as the user making an error when typing his password. However, the repeated attempts are an external attack.
In order to detect the attack, we apply the cumulative sum to the time series.

```
cumsum=S.cumsum()
S.plot()
```

Here is the cumulative sum time series:

Figure 6. The cumulative sum of failed login attempts.



Then, to detect abrupt increase in the number of failed login attempts, we compute the relative derivative of the cumulative sum.
The relative derivative of a time series x(t):

$$x'(t)=\frac{x(t)-x(t-1)}{x(t-1)}$$

To do so:

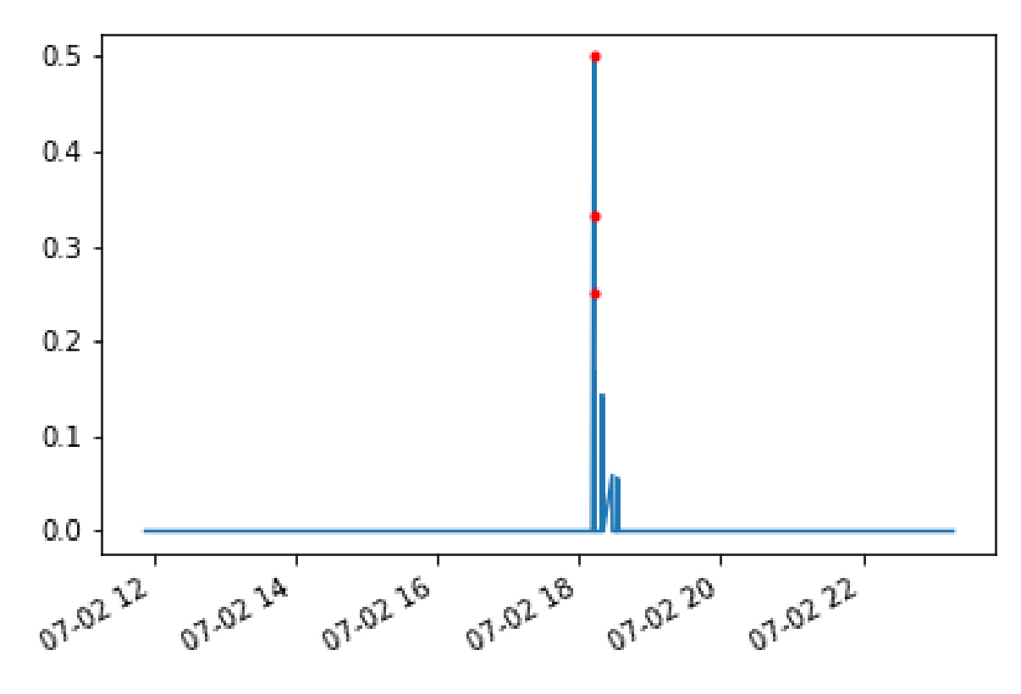
```
diff=(cumsum-cumsum.shift(periods=1))/cumsum.shift(periods=1)
diff.plot()
```

Then, we set a threshold in the derivative to trigger the alert:

```
alerts= diff.where(diff>.2).dropna()
alerts.plot(color='red', style='.')
```

We obtain the following result:

Figure 7. Relative derivative of the cumulative sum + attack detection .



This process can run in real time and be attached to a messaging system to alert the user about attacks happening in real time.
This algorithm is an example of automating authentication attempts monitoring.

auth.log contains information other than failed login attempts that could be monitored to detect security issues such as:
- session duration
- executed commands