Sistema de Detecção de Anomalia - Modelo PMA

Métodos de Autenticação e Segurança em Redes – 2021.1
Prof. Dr. Bruno Lopes Dalmazo

por
Michel Neves dos Santos

C3 – Centro de Ciências Computacionais
Universidade Federal do Rio Grande – FURG

1. Introdução

A internet está sendo ampliada cada vez mais, sua demanda como parte do cotidiano somado às mudanças tecnológicas, tanto de hardware quanto de software. Essa capacidade de processamento aumentada, junto a sistemas sofrendo constantes modificações criam brechas para ataques. Dessa forma, sistemas de detecção de intrusão estão se tornando mais recorrentes, dentre eles os modelos voltados para anomalias estão ganhando espaço devido a sua análise comportamental, eficazes na descoberta de ataques não categorizados até então. Nesse contexto, esse trabalho apresenta uma implementação de um modelo de detecção de anomalia estatístico e de abordagem local.

2. Metodologia

Foi adotado o modelo Poisson Moving Average (PMA) [1], sistema de detecção de anomalias com base estatística e análise local de dados. Esse modelo adota uma janela composta de frames, conjunto de requisições em um intervalo de tempo, e aplica o processo de Poisson truncado (ZTP) a fim de prever o próximo frame, um frame é considerado anômalo quando não se enquadra em um intervalo de limiares, além disso a versão mais recente do modelo prevê janela com dimensionamento dinâmico [2], conforme descrito na Figura 1.

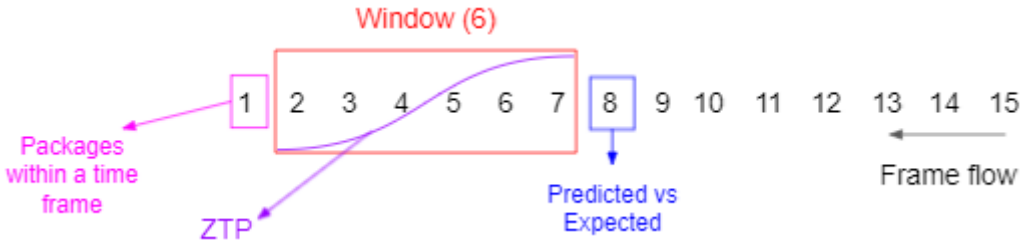

Figura 1 - Diagrama do Modelo PMA

O modelo PMA consiste em gerar uma previsão que é o somatório da multiplicação do vetor ZTP, que contém a representatividade de cada posição no formato percentual, pelo vetor janela, que possui os dados reais de cada posição.

$$prediction = \sum_{i=0}^{\lambda} window[i] \times ZTP[i]$$

A linguagem de programação utilizada foi a Python, junto aos seguintes recursos: ambiente virtual Python (venv), matplotlib, scapy, os e sys.

2.1. Venv

Venv é um recurso do Python para criar um ambiente virtual, a fim de controlar melhor as bibliotecas relacionadas ao projeto em questão.

python -m venv pasta

2.2. Matplotlib

Matplotlib é uma biblioteca utilizada para criar gráficos, ela é bem pesada e possui várias dependências.

2.3. Scapy

Scapy é uma biblioteca para manipulação e criação de pacotes de rede, foi utilizada para ler o dump de conexões. A versão estável durante o desenvolvimento apresentou problemas, dessa forma foi instalada a versão de desenvolvimento.

2.4. os

É a biblioteca padrão do Python para manipulação de arquivos, foi utilizada para remover o arquivo de dump do modelo, a fim de salvar um novo teste.

2.5. sys

É a biblioteca padrão do Python para alguns recursos do sistema, utilizada para obter o caminho do arquivo dump gerado pelo modelo.

2.6. Math

Foi utilizada a biblioteca math para cálculo exponencial e função teto.

2.7. Requirements.txt

Foi gerado um arquivo com todas dependências do projeto através do comando *pip freeze > requirements.txt*, sendo possível instalar elas de forma rápida e simples, não estou certo se funcionará da mesma forma com o scapy, pois foi instalada a versão de desenvolvimento, por isso, deixei o passo a passo no arquivo *README.md.* Para instalar as dependências, basta executar o comando abaixo.

pip install -r requirements.txt

3. Desenvolvimento

Foram desenvolvidos três códigos, um para manipular arquivos (*arquive.py)*, uma classe que implementa o modelo PMA (*model.py)* e um arquivo responsável por ler o tcpdump, filtrar pacotes, rodar o modelo, salvar os dados e mostrar resultados.

3.1. arquive.py

O arquive.py possui a classe Arquive, Figura 2, uma estrutura de manipulação de arquivos baseada no recurso *open* do Python, esse serve para ler e escrever os resultados do modelo.

```
class Arquive:
    #class constructor
    def __init__(self, file, encoding = "utf-8"):
        #save file path
        self.__file = file
        #save encoding
        self.__encoding = encoding

    #function to load all file as string
    def load(self):···

    #function to overwrite file
    def save(self, string):···

    #function to append content
    def append(self, string):···

    #function to read content split in lines
    def readAllLines(self):···
```

Figura 2 - arquive.py

A classe Arquive implementa cinco funções, o construtor que recebe o caminho para o arquivo e a codificação do conteúdo, utf-8 sendo o valor padrão. A função *load* abre o arquivo e retorna todo seu conteúdo no formato de uma string, a *save* cria se não existe ou sobreescreve o conteúdo do arquivo, o *append* cria se não existe e adiciona uma string no final do conteúdo já existente do arquivo, enquanto o *readAllLines* carrega a string do *load* e separa a string pelo marcador *\n.*

3.2. model.py

O arquivo *model.py* possui a classe PMA, que implementa o modelo PMA, ela importa apenas a biblioteca math, utilizada para cálculo exponencial e função teto.

Conforme a Figura 3, seu construtor recebe até três dados, a referência da função de alerta, chamada quando um frame é processado, o tamanho inicial da janela e a confiança da previsão do frame. Segue abaixo a descrição das variáveis da classe.

- self.__confidence_alpha - alfa da confiança do valor predito, quanto maior a confiança, maior o número de falsos positivos;
- self.__alertCallback - função de callback para cada frame;
- self.__factorial - vetor de fatorial, implementado manualmente para reduzir custo computacional;
- self.__factorial_size - tamanho do vetor fatorial;
- self.__window_size - tamanho da janela;
- self.__frame - contador de pacotes do frame;
- self.__predicted - valor predito;
- self.__poisson - vetor do processo de Poisson truncado;
- self.__time - tempo em segundos início do frame;
- self.__interval - tempo em segundos do tamanho do frame;
- self.__frames_count - contador de frames;
- self.__history_size - tamanho da lista histórico, valor padrão 100;
- self.__history - vetor histórico, uma lista FIFO;
- self.__resize_alpha - alfa de confiança para atualizar tamanho da janela, constante 0.05;
- self.__avg_old - média da janela anterior;
- self.__variance_old - variância da janela anterior;
- self.__avg - média da janela atual;
- self.__variance - variância da janela atual;

Muitos dados são preenchidos ao decorrer do código, sendo apenas inicializado no construtor por razões de boas práticas. O fatorial foi implementado manualmente, pois a fórmula de processo de Poisson truncado utiliza esse dado, tornando inviável recalcular toda uma sequência para toda mudança de valores na janela. O construtor executa as funções para calcular fatorial e vetor Poisson. A classe foi desenvolvida pensando no encapsulamento, logo apenas três funções são acessadas de fora da classe, *setStart*, *packageIn* e *stop*, descritas ao longo do trabalho.

3.2.1. adjustFactorial

A função *adjustFactorial,* Figura 3, corrige o tamanho do vetor factorial de acordo com o tamanho da janela. Se o vetor for maior, é utilizado o recurso *pop* para remover os índices que sobram, enquanto são adicionados índices no final até atingir o tamanho da janela, o novo nodo possui o valor último índice existente * tamanho do fatorial.

```python
import math

class PMA:
    def __init__(self, alertCallback, size=0, confidence = 0.92):···

    # function to set factorial vector values
    def __adjustFactorial(self):
        print("window_size", self.__window_size)
        # block to decrease factorial vector
        if(self.__factorial_size > self.__window_size):
            # decrese factorial size
            while (self.__factorial_size > self.__window_size):
                # remove last item from factorial vector
                self.__factorial.pop()
                # decrement variable that contains factorial size
                self.__factorial_size -= 1
        # block to increase factorial vector
        if(self.__factorial_size < self.__window_size):
            # increse factorial size
            while (self.__factorial_size < self.__window_size):
                # increment factorial vector with size * last item
                self.__factorial.append(
                    self.__factorial[-1] * (self.__factorial_size))
                # increment variable that contains factorial size
                self.__factorial_size += 1
```

Figura 3 - adjustFactorial.

3.2.2. adjustPoisson

A função *adjustPoisson,* Figura 4, é responsável por atualizar o vetor de poisson, ele é modificado toda vez que o tamanho da janela muda. O vetor contém a representatividade de cada índice da janela a fim de prever o próximo frame, ele é resetado e seu resultado da fórmula precisa ser normalizado. O cálculo de $e^{-\lambda}$ é realizado fora das iterações, visto que é constante para todos os índices. Segue abaixo a fórmula do processo de Poisson truncado.

$$ZTP(k) = \lambda^{k} \times e^{-\lambda} \div k!$$

```python
    # function to calculate poisson truncate distribution
    def __adjustPoisson(self):
        # reset poissson vector
        self.__poisson = []
        # pre-calculate e^-lambda
        e_m_lambda = math.exp(-self.__window_size)
        # iterate from 0 to window_size -1, ex: 0,2,3,4,5
        for k in range(self.__window_size):
            # append the result of iteration k to vector
            self.__poisson.append(math.pow(self.__window_size, k) * e_m_lambda / self.__factorial[k])

        #multiplier to normalize vector sum to 1
        normalize = 1 / sum(self.__poisson)
        #iterate poisson vector
        for i in range(self.__window_size):
            #nomalize values
            self.__poisson[i] = self.__poisson[i] * normalize
```

Figura 4 - adjustPoisson.

3.2.3. adjustWindow

A função *adjustWindow*, Figura 5, é chamada quando o algoritmo identifica ser necessário alterar o tamanho da janela, atualizando a variável de classe correspondente, então atualizando os vetores fatorial e poisson.

```
#function to resize window
def __adjustWindow(self, size):
    #set new window size
    self.__window_size = size
    #recalculate factorial
    self.__adjustFactorial()
    #recalculate poisson truncate distribution
    self.__adjustPoisson()
```

Figura 5 - adjustWindow.

3.2.4. predict

A função *predict*, Figura 6, implementa o somatório a seguir:

$$prediction = \sum_{i=0}^{\lambda} window[i] \times ZTP[i]$$

```
# generate prediction
def __predict(self):
    # reset predicted
    self.__predicted = 0
    # iterate from 0 to window_size - 1
    for i in range(self.__window_size):
        # accumulate frame relevance * frame
        self.__predicted += (
            self.__poisson[i] * self.__history[self.__history_size - self.__window_size + i])
```

Figura 6 - Predict.

Conforme visualizado, é utilizada a lista FIFO histórica para armazenar os frames ao invés de uma janela exclusiva para os pertinentes, já que a janela pode mudar de tamanho, é necessário preencher com os frames antigos, por tanto, tudo foi feito com base no mesmo vetor. A janela então é indicada pela variável *self.__window_size*, representando o tamanho da janela, dessa forma, o primeiro índice presente é o tamanho do histórico - tamanho da janela, enquanto o último é o tamanho da janela - 1.

3.2.5. calcVariance

A função calcVariance, Figura 7, é utilizada para calcular a variância, necessária para ajustar o tamanho da janela.

$$\sigma^2 = \sum_{k=0}^{\lambda} (window[k] - \mu)^2 \div \lambda$$

```
# function to calculate variance
def __calcVariance(self):
    #print("__________calcVariance__________")
    # reset current variance
    self.__variance = 0
    # iterate window frames
    for frame in self.__history[self.__history_size - self.__window_size:]:
        # sommation
        self.__variance += math.pow(frame - self.__avg, 2)
        #print(self.__variance)
    # divide summation by vector size
    self.__variance = self.__variance / self.__window_size

    #print("__________end calcVariance__________")
```

Figura 7 - calcVariance.

3.2.6. calcMinMax

A função calcMinMax, Figura 8, calcula o menor e maior valores da janela, recursos necessários para o ajuste do tamanho da janela.

```
# function to get min and max frame from window
def __calcMinMax(self):
    # set min_frame as infinite
    min_frame = float("inf")
    # set max_frame as zero
    max_frame = 0
    # iterate all frames from window
    for frame in self.__history[self.__history_size - self.__window_size:]:
        # save frame if it's lower than min_frame
        min_frame = (frame < min_frame) * frame or min_frame
        # save frame if it's higher than max_frame
        max_frame = (frame > max_frame) * frame or max_frame
    # return min and max frames
    return min_frame, max_frame
```

Figura 8 - calcMinMax.

3.2.7. slideWindow

A função *slideWindow*, Figura 9, é uma das funções mais complexas e importantes da classe, responsável pelo deslizamento da janela, bem como pelo redimensionamento da mesma. Ela salva os valores de variância e média nas variáveis de backup (*_old*), adiciona o novo frame no final do histórico e remove o mais antigo, mantendo o mesmo tamanho de vetor ao final das operações. Após atualizar o vetor histórico, é necessário recalcular a média e variância a fim de verificar a necessidade de ajustar o tamanho da janela.

Utilizando a fórmula apresentada em [2], é feito o reajuste se pertinente e então a função teto transforma o tamanho float em um dado inteiro, visto que a janela é uma lista e como tal não existe posição float, finalizando ao executar a função *adjustWindow*.

```
# function to slide window and calculate statistics
def __slideWindow(self):
    # save current variance as old
    self.__variance_old = self.__variance
    # save current avg as old
    self.__avg_old = self.__avg
    # append frame to history
    self.__history.append(self.__frame)
    # remove oldest frame from history
    self.__history.pop(0)
    # calculate new avg
    self.__avg = sum(
        self.__history[self.__history_size - self.__window_size:])/self.__window_size
    # calculate new variance
    self.__calcVariance()
    #check if window is not empty
    if(self.__avg != 0 and self.__avg_old != 0):
        direct = self.__avg / self.__avg_old
        inverse = self.__avg_old / self.__avg
        ratio = abs(direct - inverse)

        # verify whether ration above threshold
        if(ratio >= 1 + self.__resize_alpha):
            # get min and max frame from window
            min_frame, max_frame = self.__calcMinMax()
            # calculate max variance
            variance_max = (self.__avg - min_frame) * \
                (max_frame - self.__avg)
            # calculate volume to resize window
            volume = variance_max / self.__variance
            # calculate new window size
            new_size = self.__window_size + \
                ((self.__avg > self.__avg_old) * volume or -volume)

            # call adjust window function passing ceil of new size
            self.__adjustWindow(math.ceil(new_size))
```

Figura 9 - slideWindow

3.2.8. anomaly

Anomaly é a função que verifica se uma anomalia encontrada, um frame pode ser classificado em quatro categorias, anomalia alta ou baixa, valor normal ou em treinamento. Um frame é considerado em treinamento quando a janela não foi completamente preenchida,

normal quando preenchida e no intervalo de confiança (padrão 92%), anomalia abaixo do normal quando o valor não atinge o limite inferior, e alta quando ultrapassa o limite superior. O alerta é gerado por frame, sendo passados a categoria do frame, número de pacotes no seu intervalo, valor predito, timestamp inicial e duração do frame.

```
# function to check anomaly
def __anomaly(self):
    # verify if it's above threshold
    if(self.__frames_count >= self.__window_size and self.__frame > self.__predicted * (1 + self.__confidence_alpha)):
        # alert as higher frame, sending frame and predicted
        self.__alertCallback("higher", self.__frame, self.__predicted,self.__time,self.__interval)
    # verify if it's below threshold
    elif (self.__frames_count >= self.__window_size and self.__frame < self.__predicted * (1 - self.__confidence_alpha)):
        # alert as lower frame, sending frame and predicted
        self.__alertCallback("lower", self.__frame, self.__predicted,self.__time,self.__interval)
    # verify if it's in training
    elif (self.__frames_count < self.__window_size):
        # alert as training frame, sending frame and predicted
        self.__alertCallback("training", self.__frame, self.__predicted,self.__time,self.__interval)
    # frame within prediction
    else:
        # alert as normal frame, sending frame and predicted
        self.__alertCallback("normal", self.__frame, self.__predicted,self.__time,self.__interval)
```

Figura 10 - anomaly

3.2.9. setStart

A função pública *setStart,* Figura 11, serve para setar os valores de timestamp do início do primeiro frame e duração dos frames ao longo do teste.

```
# set initial value for time and interval
def setStart(self, start_time, interval):
    # set variable time
    self.__time = start_time
    # set time interval
    self.__interval = interval
```

Figura 11 - setStart

3.2.10. packageIn

A função pública *packageIn*, Figura 12, é responsável por receber o timestamp do pacote e se ele não estoura o limite de tempo do frame, ele é contabilizado como parte do frame, caso contrário, é realizado o teste de anomalia, a janela é deslizada, o contador de frames é incrementado, o timestamp do próximo frame é ajustado e o frame é resetado, recebendo o valor 1 (um).

```
# function to processo package in
def packageIn(self, time):
    # verify if package time is out of frame time
    if(time > self.__time + self.__interval):
        # check frame anomaly
        self.__anomaly()
        # recalculate window and slide window
        self.__slideWindow()
        # set next frame base time
        self.__time += self.__interval
        # frames count increment
        self.__frames_count += 1
        # start new frame count
        self.__frame = 1
    # increment package in frame
    self.__frame += 1
```

Figura 12 - packageIn.

3.2.11. stop

A função pública *stop*, Figura 13, é executada quando os pacotes acabam, encerrando o frame após incrementar o contador de frames, realizar os testes de anomalia e deslizar a janela.

```
# function to end model analysis
def stop(self):
    # sum 1 to frames count
    self.__frames_count += 1
    # call anomaly test
    self.__anomaly()
    # recalculate window and slide window
    self.__slideWindow()
    # start new frame count
    self.__frame = 1
```

Figura 13 - stop

3.3. main.py

O arquivo *main.py* possui o processamento do tcpdump, formato das requisições, salva os resultados e gera dois gráficos. Ele possui alguns imports de bibliotecas de sistema, processamento de pacotes, recursos gráficos, classes de modelo PMA e de manipulação de arquivos. O arquivo main.py possui três funções além do código de configuração abaixo.

```
if ("__main__" == __name__):
    #check if no file is passed as arg
    if(len(sys.argv) <= 2):
        #check if confidence is not an arg
        if(len(sys.argv) == 1):
            #append confidence
            sys.argv.append(0.1)
```

```
    else:
        #cast confidence to fload
        sys.argv[1] = float(sys.argv[1])
    #append default file as sys args
    sys.argv.append("PMA_result.txt")
#check if file already exists
if os.path.exists(sys.argv[2]):
    #delete file
    os.remove(sys.argv[2])
# file to test
file = "Data/week1/monday.tcpdump"
# process file
pcap(file)
#show graphs
showResults()
```

Esse código verifica se foi passado a confiança no valor predito e o arquivo como parâmetro para salvar o resultado do modelo, caso contrário é utilizado 0.1 e "PMA_result.txt" como valores padrão, então ele é deletado caso exista no diretório. o *file* contém o caminho para o backup a ser lido, executando a função de processamento de captura e mostrando os gráficos do resultado.

3.3.1. frameFeedBack

É a função responsável por salvar o retorno do alerta, Figura 14, recebendo o status do frame, o valor do frame, valor predito, timestamp do início do frame e duração. Os dados são colocados numa string pelo separador vírgula e adicionados ao arquivo de resultados, o caminho para o arquivo fica no índice um do vetor de argumentos do programa.

```
# function to process frame status

def frameFeedback(status, frame, predicted, time, interval):
    file = sys.argv[1]
    #class to handle file connection
    arq = Arquive(file)
    #append data to file
    arq.append(f'{status},{frame},{predicted},{time},{interval}\n')
```

Figura 14 - frameFeedback.

3.3.2. pcap

A função *pcap* é responsável por iniciar o modelo e carregar o arquivo recebido, processar os pacotes, filtrando apenas o protocolo UDP, verificando desde Ethernet, IPV4 até o UDP propriamente dito. Quando o primeiro pacote válido é lido, são setados os valores iniciais de

tempo de frame para o modelo, função *setStart*, então é utilizada a função *packageIn* passando o metadado timestamp do pacote. Ao término dos pacotes, é invocada a função *stop* para que o frame em contagem seja dado como encerrado.

```
def pcap(file_name):
    print('Opening {}...'.format(file_name))
    # size of window
    size = 10
    #interval in secs
    interval = 60
    # initialize model
    pma = PMA(frameFeedback, size, sys.argv[1])
    # package counter
    count = 0
    # iterate in all packages as a vector of classes
    for (pkt_data, pkt_metadata,) in RawPcapReader(file_name):
        #Ethernet parser
        ether_pkt = Ether(pkt_data)
        #check if it's LLC protocol
        if 'type' not in ether_pkt.fields:
            #skip package
            continue
        # ignore non-IPv4 packets
        if ether_pkt.type != 0x0800:
            #skip package
            continue
        #parse IPV4 package
        ip_pkt = ether_pkt[IP]
        # Ignore non-UDP packet
        if ip_pkt.proto != 17:
            # skip package
            continue
        # increment counter
        count += 1
        # verify if is first iteration
        if count == 1:
            # set model start time
            pma.setStart(pkt_metadata.sec, interval)
        # send package time to model
        pma.packageIn(pkt_metadata.sec)
    # end model
    pma.stop()
```

3.3.3. showResults

A função *showResults*, tem como finalidade mostrar os resultados de forma gráfica, um gráfico de linhas e outro de dispersão, o primeiro apresenta a projeção dos valores de frame e predição em função do número do pacote, enquanto o segundo disponibiliza a projeção dos frames com cores indicando sua classificação.

São criados vetores para armazenar cada tipo de informação proveniente do arquivo de resultados, em frames, predições, status, cores e contagem de frames. A partir desses vetores são gerados os gráficos de linha e de dispersão.

- Linha:
  - Azul - valor real do frame
  - Verde claro - valor predito
- Dispersão:
  - Vermelho - acima do limite
  - Amarelo - abaixo do limite
  - Verde - normal
  - Preto - frame de treinamento

4. Resultados

Os resultados foram muito interessantes, uma implementação funcional que utiliza minimamente os recursos disponíveis na internet. É possível modificar alguns parâmetros na execução do código, tais como arquivo de resultado e confiança no valor predito, é recomendado utilizar valores baixos para a confiança, pois não é um dataset com estabilidade no volume de requisições. Como padrão, foi adotado um minuto como intervalo de frames, pois a quantidade de requisições varia muito no intervalo de tempo curto.

A quantidade de alertas abaixo do limiar foi de 5 (cinco), enquanto os acima do limiar foi de 94 (noventa e quatro), Figura 15, para o arquivo do dataset do MIT de 1999, segunda-feira da semana quatro, boa parte dos alertas condiz com os horários dos ataques, de forma que apenas 24,24% dos alertas foram em horários de estabilidade, logo 75,76% dos alertas foram em momentos duvidosos, pois o intervalo entre um alerta e outro é consideravelmente grande.

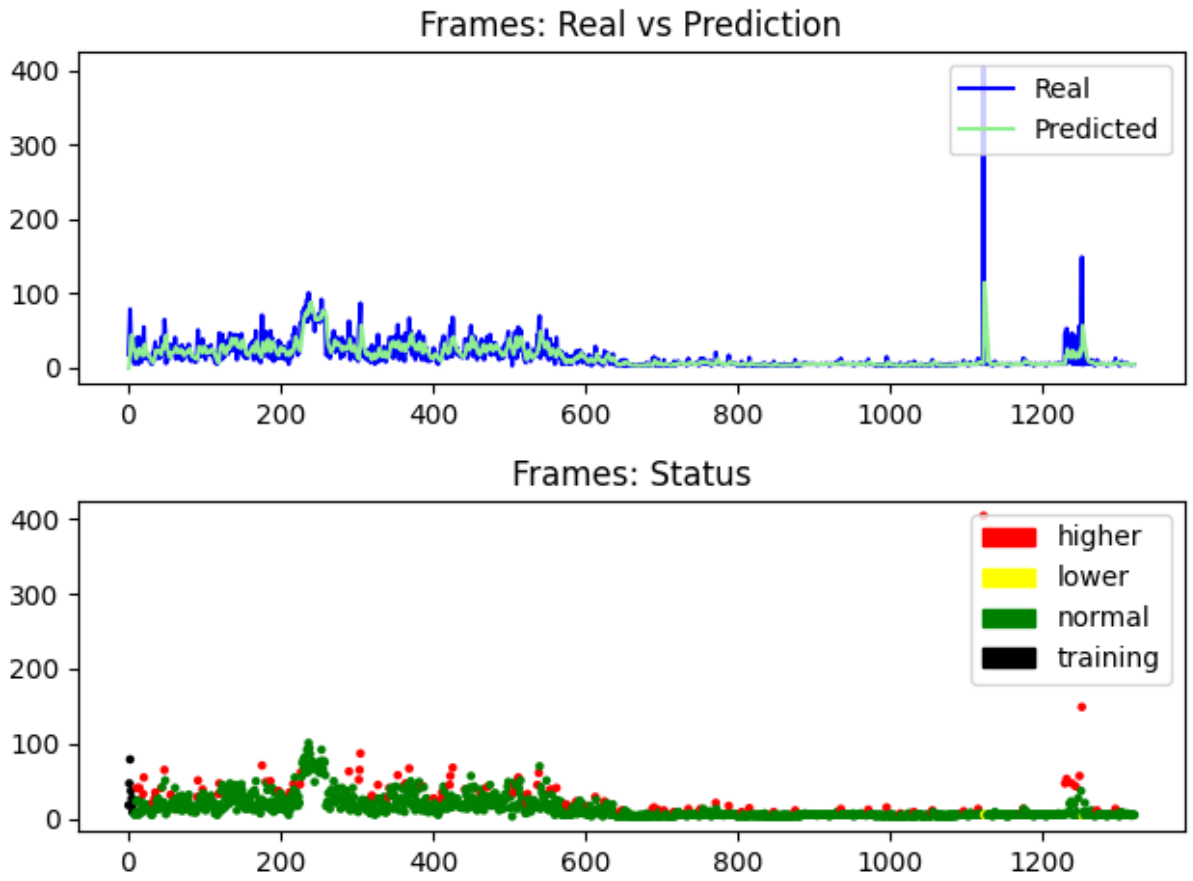

Figura 15 - Gráficos segunda-feira, semana quatro

5. Conclusões

Conclui-se que o modelo PMA é bastante eficiente para detectar anomalias volumétricas, prevê a dinamicidade da rede e é um modelo estatístico, portanto tende a ser mais rápido e consumir menos recursos. O único ponto negativo é a dificuldade de parametrizar o limiar de anomalia e a duração dos frames.

6. Links

Link para o vídeo: https://youtu.be/osHBmUEgBWI
Link para o código: https://github.com/nevessmichel/AnomalyPMA

7. Referências

[1] DALMAZO, B. L.; VILELA, J. P.; CURADO, M. Predicting traffic in the cloud: A Statistical approach. *In:2013 International Conference on Cloud and GreenComputing. [S.l.: s.n.]*, 2013. p. 121–126.

[2] DALMAZO, B. L.; VILELA, J. P.; CURADO, M. Online traffic prediction in the cloud. *Int. J. Network Mgmt*, 2016, 26: 269– 285.