Language Model based Detection Approach of Algorithmically Generated Malicious Domain Names

Xiaodan Li

Introduction:

With the increasing importance of network, network security becomes more and more demanding in almost every aspect. In 2015, IBM and Ponemon Institute conducted research on the cost due to data breach in 62 companies. The average cost of data breach is $6.5 million [1]. In order to break into target system and make more damage, the infected hosts need to connect back to command & control (C2) by domain name generation algorithms to get more commands or payload. DGA generates a large number of domain names randomly which can be meeting points with their C2 servers. The domain name is generated randomly and periodically. There are many domain generation algorithm (DGA), such as conficker, cryptolocker, ramdo and so on. Those algorithms generate domains pseudo-randomly by the seed which is shared by malware and threat actor. The threat actor knows the sequence of connection by malware. Therefore, it is really hard to detect it. As to the attacker, the economics are really good, while it is cost intensive for defender. The attackers only need to generate the domain names and register them if they will be used in the attacking process. However, as to the defenders, they need to figure out all of the possible domains and put them into blacklist. The security practitioners need to reverse engineer the DGA. This is so time- and resource- intensive. Even if the defenders figure out what exactly the algorithm is, the attackers can modify the original code to have a new group of domains [2]. In this project, three different language models: perplexity of bigram language model, K-L distance with bigram distribution and K-L distance with unigram distribution, are applied to detect random generated malicious domain names. We only focus on the domain name without contextual information. In addition, we do not need to figure out how the DGA works and the time to detect DGA is fast since all of the analysis is based on the domain name itself. The assumption of this approach is that DGA tends to not use the well-formed and pronounceable language words since the domains are generated randomly. Most of the registered domains are constructed by the legitimate words. Therefore, the DGA domain names will use characteristics with a different distribution as the legitimate domain names.

Methods:

a. Perplexity of bigram language model

In this model, firstly we will train the bigram matrix by benign data set. In this matrix, every column and row represent one character. The row represents the former character while row stands for following character. We train every benign domain name and find bigrams in the string. After that, we will have a matrix with every element as the count of combination of characters pointed by row and column. However, there are so many zero entries in the matrix. In order to smooth the matrix and move a little bit probability mass from frequent bigrams to rare bigrams, add-one smooth is applied. Then the probability matrix is transformed into log probability to avoid numeric underflow. As to the testing domain name, we will compute its perplexity. The perplexity is the metric to determine whether the current domain name is DGA or not. Perplexity is related to

probability that a string occurs [3]. According to our assumption, there will be a vast difference of perplexity between benign domains and DGA.

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \tag{1}$$

As to the dataset, the benign training dataset is the 980000 domain names from Amazon Alexa top-1m global site. The Amazon Alexa includes the top 1 million common website domain names. The testing set includes 20000 domain names from Alexa and malicious domain names from different DGA generators: conficker, cryptolocker, opendns-random-domains, zeus, tinba, rovnix, ramdo, pushdo. Before training the benign domain names, we need preprocess the website string. firstly, we need to delete the TLD. Then the initial character should be '^', while the ending character should be '$'.

b. K-L distance with unigram distribution

K-L (Kullback-Leibler) distance is used to measure the distance between to distribution. We can use metric to determine which distribution the current value is closer to. Therefore, we will have two distributions. One is the character distribution for benign domains, while another one is the character distribution for DGA [2].

$$P(\underline{x} \mid \underline{g}) \underset{b}{\overset{g}{\gtrless}} P(\underline{x} \mid \underline{b}). \tag{2}$$

As shown in equation (2), x is the domain name we need to classify. g is the character distribution of benign domain name. b is the character distribution of malicious domain name [2].

$$\begin{aligned} P(\underline{x} \mid \underline{g}) &= \prod_{k=1}^{N} P(x_k \mid \underline{g}) = \prod_{i=1}^{M} P(a_i \mid \underline{g})^{n_i} \\ &= \prod_{i=1}^{M} g_i^{n_i} = \prod_{i=1}^{M} g_i^{q_i N}. \end{aligned} \tag{3}$$

According to equation 3, we can get P(x|g) which is the probability conditioned on benign character distribution. g_i is the character probability from benign distribution. n_i is the occurrence of ith character [2].

$$\begin{aligned} P(\underline{x} \mid \underline{b}) &= \prod_{k=1}^{N} P(x_k \mid \underline{b}) = \prod_{i=1}^{M} P(a_i \mid \underline{b})^{n_i} \\ &= \prod_{i=1}^{M} b_i^{n_i} = \prod_{i=1}^{M} b_i^{q_i N}. \end{aligned} \tag{4}$$

We can get P(x|b) which is probability conditioned on malicious distribution. b_i is the character probability from malicious distribution. n_i is the occurrence of ith character [2].

As to K-L distance with bigram distribution, firstly, we will summarize the unigram character distribution for benign domain distribution and malicious domain distribution. The former 980000 domain names from Amazon Alexa top-1m global site is used to get benign unigram character distribution. The former 80000 domain names from conficker are used to get the malicious unigram character distribution. The remaining domain names are the testing set. As to every testing

record, we will compute probability conditioned on good domain name distribution and malicious name distribution separately which are shown in equation 2. Then if the probability conditioned on legitimate domain name distribution is larger than that conditioned on malicious domain name distribution. The current domain name is benign. Otherwise, the current domain name is malicious.

c. K-L distance with bigram distribution

The principle for bigram distribution is similar to unigram. The two consecutive characters are considered in this metric. The benign dataset is the former 980000 domain names from Amazon Alexa top-1m global site. The former 80000 domain names from conficker are used to train malicious bigram character distribution. If the character pair does not occur, the probability is assign as 10e-6. As to every tested domain name, we will get its bigrams. Then get probability conditioned on benign and malicious character distributions.

Results and analysis:

a. Results of perplexity of bigram language model

Table 1 shows the different statistics about perplexity such as mean, min and max for each dataset. From this table, we can find that the difference between benign dataset and malicious datasets are obvious. Therefore, we can use a boundary to classify benign domain names and DGA easily. The legitimate domain usually consists of meaningful words, while the combination of characters by DGA is random. The distribution of characters will be so different. Therefore, the perplexity is various. According to the table, we can find out that except for Zeus, the perplexity of DGA domain name is close. Those are around 4.5. However, the perplexity ranges are so different.

Table 1 Perplexity statistics of different domain name dataset

	Alexa	Conficker	Crypto-locker	Opendns-random-domains	Zeus	Tinba	Rovnix	Ramdo	Pushdo
Mean	18.29	4.39	4.57	4.06	6.58	4.27	4.24	4.65	4.29
Min	15.75	2.83	3.15	2.20	3.29	3.15	3.27	3.22	3.02
Max	19.62	6.87	7.03	9.53	9.89	6.92	8.94	7.05	6.67

b. Result of K-L distance

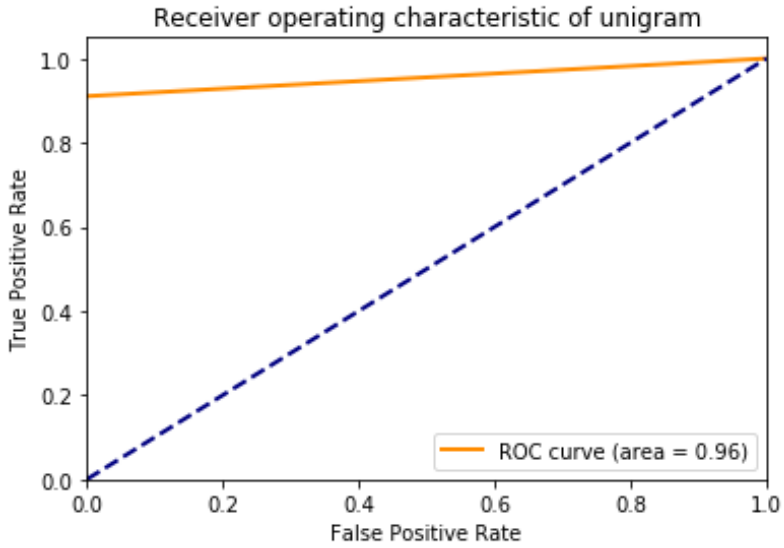

Figure 1. ROC for K-L distance with unigram distribution (conflicker)

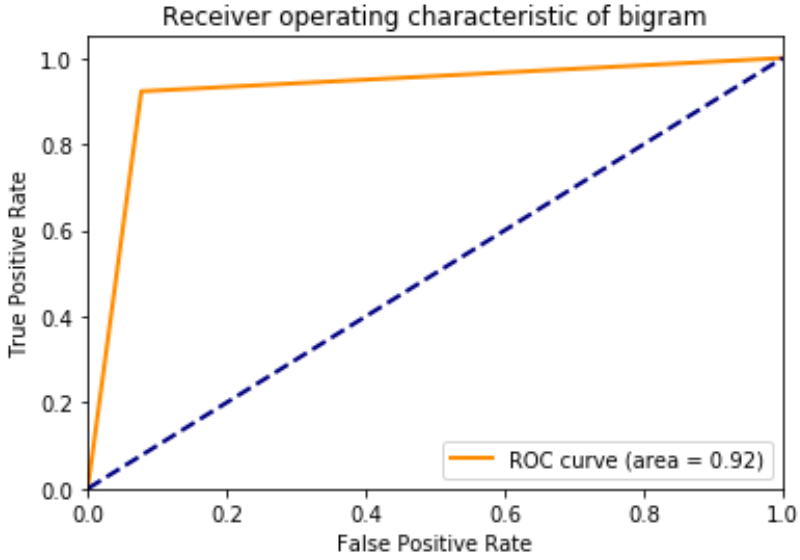

Figure 2. ROC for K-L distance with bigram distribution (conflicker)

In the figure 1 and 2, we can find out that K-L distance with unigram distribution / bigram distribution can classify DGA and validate domain name. DGA from conflicker is used as the malicious domain names. What surprise us is that K-L distance with unigram distribution outperforms K-L distance with bigram distribution. There are some reasons. All of the unigram probabilities are obtained from training data directly, while there might be some bigrams which do not show up in the training set, so the probability for the bigram which does not occur is not accurate. The probability assigning to rare bigram is 10-06, which is the random number.

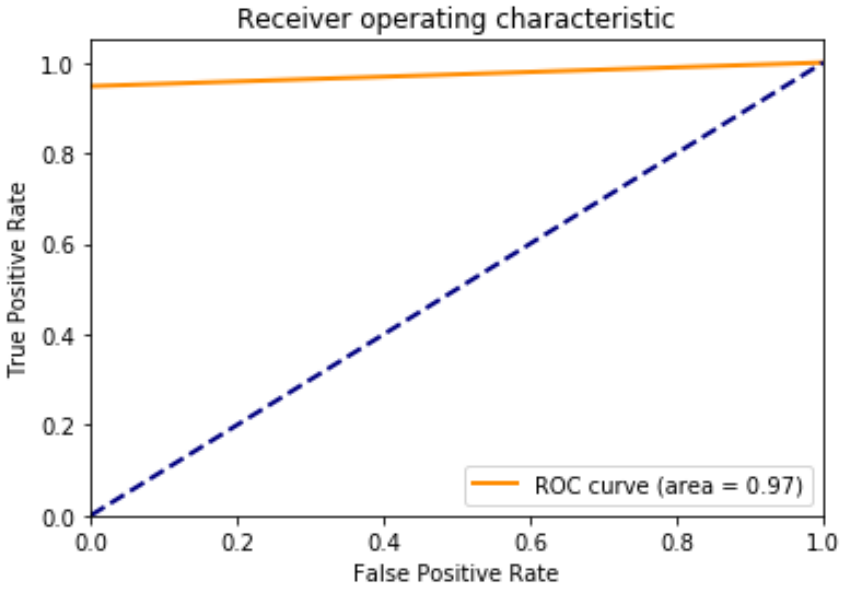

Figure 3. ROC for K-L distance with unigram (zeus)

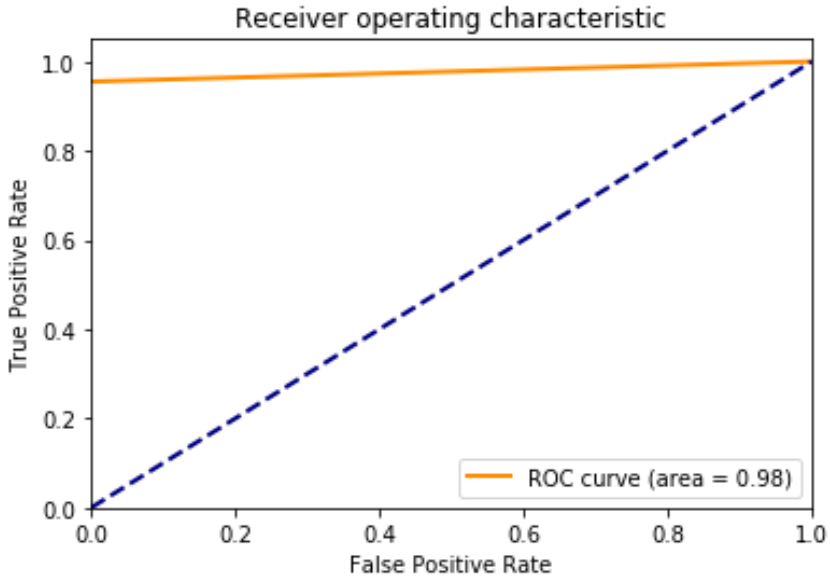

Figure 4. ROC for K-L distance with bigram (zeus)

In the figure 3 and 4, we can get the classification performance with unigram / bigram on zeus dataset. In this case, both of models have a good performance. The bigram model has a slightly better performance than unigram model. This result is opposite to what we get from conflicker. When we compare the algorithms to generate domain name and DGA, we can find out conflicker generates domain name with shorter length than zeus. It is more difficult to detect random domain name from conflicker.

In general, after comparing perplexity and K-L distance, we can find out if we select a proper perplexity value which can perform much better than K-L distance. This is because when training bigram model for perplexity, as to zero entries, add one smooth is used instead of assigning value like 10-6 to bigram. Therefore, the distribution of probability will be more reasonable. However,

as to perplexity, we only set up a fixed number which might be not flexible if there are other DGA algorithms. K-L distance can overcome this problem efficiently.

Reference:

[1] https://www-03.ibm.com/press/us/en/pressrelease/47022.wss
[2] Yadav, Sandeep, Ashwath Kumar Krishna Reddy, A. L. Reddy, and Supranamaya Ranjan. "Detecting algorithmically generated malicious domain names." In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pp. 48-61. ACM, 2010. Harvard.
[3] Martin, James H., and Daniel Jurafsky. "Speech and language processing." International Edition 710 (2000): 25.

```
import pandas as pd
import numpy as np
import math
import NLP_module

df = pd.read_csv('/Users/lixiaodan/Desktop/ece590/DGA/top-1m.csv', header=None)
goodNames = df[1]
gDomains = list()
for i in range(len(goodNames)):
    domainTp = goodNames[i].split('.')[0]
    gDomains.append('^'+ domainTp + '$')
trainGDom = gDomains[0:980000]
testGDom = gDomains[980000:]

bunigram = NLP_module.getUnigram(gDomains)

chars = list(bunigram.keys())
biMatrix = NLP_module.getBiProMatrix(bunigram, chars, gDomains)

logMatrix = np.zeros((len(chars), len(chars)))
for i in range(len(chars)):
    for j in range(len(chars)):
        logMatrix[i][j] = math.log2(biMatrix[i][j])

gPPWs = NLP_module.getPPWs(trainGDom, logMatrix, chars)
goodMean = np.mean(gPPWs)
print("PPW for valid domain name")
print(goodMean)
print(np.min(gPPWs))
print(np.max(gPPWs))

url = '/Users/lixiaodan/Desktop/ece590/DGA/conficker.txt'
print("PPW for conficker")
NLP_module.printStaticsPPW(url, logMatrix, chars)

url = '/Users/lixiaodan/Desktop/ece590/DGA/cryptolocker.txt'
print("PPW for cryptolocker")
NLP_module.printStaticsPPW(url, logMatrix, chars)

url = '/Users/lixiaodan/Desktop/ece590/DGA/opendns-random-domains.txt'
print("PPW for opendns-random-domains")
NLP_module.printStaticsPPW(url, logMatrix, chars)

url = '/Users/lixiaodan/Desktop/ece590/DGA/zeus.txt'
print("PPW for zeus")
NLP_module.printStaticsPPW(url, logMatrix, chars)
```

```
url = '/Users/lixiaodan/Desktop/ece590/DGA/tinba.txt'
print("PPW for tinba")
NLP_module.printStaticsPPW(url, logMatrix, chars)

url = '/Users/lixiaodan/Desktop/ece590/DGA/rovnix.txt'
print("PPW for rovnix")
NLP_module.printStaticsPPW(url, logMatrix, chars)

url = '/Users/lixiaodan/Desktop/ece590/DGA/ramdo.txt'
print("PPW for ramdo")
NLP_module.printStaticsPPW(url, logMatrix, chars)

url = '/Users/lixiaodan/Desktop/ece590/DGA/pushdo.txt'
print("PPW for pushdo")
NLP_module.printStaticsPPW(url, logMatrix, chars)


import pandas as pd
import NLP_module

df = pd.read_csv('/Users/lixiaodan/Desktop/ece590/DGA/top-1m.csv', header=None)
goodNames = df[1]
gDomains = list()
for i in range(len(goodNames)):
    domainTp = goodNames[i].split('.')[0]
    gDomains.append(domainTp)

trainGDom = gDomains[0:980000]
testGDom = gDomains[980000:]

# train benigh letters distibution to get bunigram
bunigramPro = NLP_module.getUnigramPro(trainGDom)

# train benigh strings to get bbigram
bbigramPro = NLP_module.getBigramPro(trainGDom)

# get malicious letters distribution
#file = open('/Users/lixiaodan/Desktop/ece590/DGA/conficker.txt', 'r')
file = open('/Users/lixiaodan/Desktop/ece590/DGA/zeus.txt', 'r')
dgas = list()
for line in file:
    dga = line.split('.')[0]
    dgas.append(dga)

mtotal = 0
```

```
trainDgas = dgas[0:80000]
testDgas = dgas[80000:]

munigramPro = NLP_module.getUnigramPro(trainDgas)

# train malicious strings to get mbigram
mbigramPro = NLP_module.getBigramPro(trainDgas)

# 1 - dga, 0 - valid domain
# testing benign dataset
gDomByUni = list()
gDomByBi = list()
for gdDom in testGDom:
    KLdisUni = NLP_module.getUniKLdis(bunigramPro, munigramPro, gdDom)
    if KLdisUni <= 0:
        gDomByUni.append(1)
    else :
        gDomByUni.append(0)

    KLdisBi = NLP_module.getBiKLdis(bbigramPro, mbigramPro, gdDom)
    if KLdisBi <= 0:
        gDomByBi.append(1)
    else:
        gDomByBi.append(0)

dgaByUni = list()
dgaByBi = list()
for dga in testDgas:
    KLdisUni = NLP_module.getUniKLdis(bbigramPro, mbigramPro, dga)
    if KLdisUni <= 0:
        dgaByUni.append(1)
    else :
        dgaByUni.append(0)

    KLdisBi = NLP_module.getBiKLdis(bbigramPro, mbigramPro, dga)
    if KLdisBi <= 0:
        dgaByBi.append(1)
    else:
        dgaByBi.append(0)

y_label = [0] * 20000 + [1] * 20000
resultByUni = gDomByUni + dgaByUni
resultByBi = gDomByBi + dgaByBi

NLP_module.plotRoc(resultByUni, y_label)
NLP_module.plotRoc(resultByBi, y_label)
```

```
import numpy as np
import math
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# get the K-L divergence for unigram
def getUniKLdis(bunigram, munigram, inputStr):
    KL_dis = 0
    N = len(inputStr)
    pxg = 1
    for c in inputStr:
        if c in bunigram.keys() and bunigram.get(c) != None:
            pxg = pxg * bunigram.get(c)
        else:
            pxg = pxg * 10e-6

    # get P(x|b)
    pxb = 1
    for c in inputStr:
        if c in munigram.keys() and munigram.get(c) != None:
            pxb = pxb * munigram.get(c)
        else:
            pxb = pxb * 10e-6

    if KL_dis == 0:
        KL_dis = 1.0 / N * math.log(pxg * 1.0 / pxb)
    return KL_dis

# get Kullback-Liebler distance based on bigram
def getBiKLdis(bbigram, mbigram, inputStr):
    KL = 0
    pxg = 1
    N = len(inputStr)
    # get pxg
    tp = ""
    for i in range(len(inputStr)):
        tp = tp + inputStr[i]
        if i >= 1:
            if tp in bbigram.keys() and bbigram.get(tp) != None:
                pxg = pxg * bbigram.get(tp)
            else:
                pxg = pxg * 10e-6
            tp = tp[1:]

    # get pxb
    pxb = 1
```

```
    tp = ""
    for i in range(len(inputStr)):
        tp = tp + inputStr[i]
        if i >= 1:
            if tp in bbigram.keys() and mbigram.get(tp) != None:
                pxb = pxb * mbigram.get(tp)
            else:
                pxb = pxb * 10e-6
            tp = tp[1:]

    if KL == 0:
            KL = 1.0 / N * math.log(pxg * 1.0 / pxb)
    return KL

def plotRoc(y_predict, y_label):
    # roc for unigram
    fprUni, tprUni, _ = roc_curve(y_predict, y_label)
    roc_aucUni = auc(fprUni, tprUni)

    plt.figure()
    lw = 2
    plt.plot(fprUni, tprUni, color='darkorange',
             lw=lw, label='ROC curve (area = %0.2f)' % roc_aucUni)
    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic')
    plt.legend(loc="lower right")
    plt.show()

def getUnigramPro(dataset):
    bunigram = dict()
    totalCh = 0
    for domain in dataset:
        totalCh = totalCh + len(domain)
        for char in domain:
            if char in bunigram.keys():
                bunigram[char] = bunigram.get(char) + 1
            else:
                bunigram[char] = 1
    for key in bunigram.keys():
        bunigram[key] = bunigram[key] * 1.0 / totalCh
    return bunigram
```

```
def getUnigram(dataset):
    bunigram = dict()
    for domain in dataset:
        for char in domain:
            if char in bunigram.keys():
                bunigram[char] = bunigram.get(char) + 1
            else:
                bunigram[char] = 1
    return bunigram

def getBigramPro(dataset):
    bbigram = dict()
    tt = 0
    n = 2
    for domain in dataset:
        if len(domain) <= 1:
            continue
        tt = tt + len(domain) - (n - 1)
        tp = ''
        for i in range(len(domain)):
            tp = tp + domain[i]
            if i >= (n - 1):
                if tp in bbigram.keys():
                    bbigram[tp] = bbigram.get(tp) + 1
                else :
                    bbigram[tp] = 1
                tp = tp[1:]
    for key in bbigram.keys():
        bbigram[key] = bbigram[key] * 1.0 / tt
    return bbigram

def getBigram(dataset):
    bbigram = dict()
    n = 2
    for domain in dataset:
        if len(domain) <= 1:
            continue
        tp = ''
        for i in range(len(domain)):
            tp = tp + domain[i]
            if i >= (n - 1):
                if tp in bbigram.keys():
                    bbigram[tp] = bbigram.get(tp) + 1
                else :
                    bbigram[tp] = 1
                tp = tp[1:]
```

```
    return bbigram

def getBiList(dataset):
    biList = dict()
    tt = 0
    n = 2
    for domain in dataset:
        if len(domain) <= 1:
            continue
        tt = tt + len(domain) - (n - 1)
        tp = ''
        curBis = list()
        for i in range(len(domain)):
            tp = tp + domain[i]
            if i >= (n - 1):
                curBis.append(tp)
                tp = tp[1:]
        biList[domain] = curBis
    return biList

def getPPWs(testSet, logMatrix, rowNames):
    PPWs = list()
    for test in testSet:
        testStr = '^'+ test + '$'
        N = len(testStr)
        PPW = 1
        cur = ''
        for i in range(N):
            cur = cur + testStr[i]
            if i >= 1:
                row = rowNames.index(cur[0])
                col = rowNames.index(cur[1])
                PPW = PPW + logMatrix[row][col]
        PPW = PPW * (-1.0) / N
        PPWs.append(PPW)
    return PPWs

def printStaticsPPW(url, biMatrix, rowNames):
    file = open(url, 'r')
    PPWs = list()
    for line in file:
        tp = line.split('.')[0]
        PPWs.append(tp)
    PPW = getPPWs(PPWs, biMatrix, rowNames)
    print(np.mean(PPW))
    print(np.min(PPW))
```

```
    print(np.max(PPW))

def getBiProMatrix(bunigram, chars, gDomains):
    biMatrix = np.zeros((len(chars), len(chars)))
    for domain in gDomains:
        cur = ''
        for i in range(len(domain)):
            cur = cur + domain[i]
            if i >=1:
                row = chars.index(cur[0])
                col = chars.index(cur[1])
                biMatrix[row][col] = biMatrix[row][col] + 1
                cur = cur[1:]
    # add v smoothing
    for i in range(len(chars)):
        cwn_1 = bunigram.get(chars[i])
        for j in range(len(chars)):
            biMatrix[i][j] = ( biMatrix[i][j] + 1 ) * 1.0 / (cwn_1 + len(chars))
    return biMatrix
```