**Detecting traffic type using machine learning algorithms**

Typically, detecting traffic type requires utilizing deep packet inspection approaches. In this article we take a different approach and try to utilize machine learning techniques instead: that is without looking at packet port numbers and other protocol fields, but rather only looking at traffic statistics we try to predict the type of OSI L7 application. Essentially, we are going to use features that are extracted from the TCP and UDP flows to detect type of application being used. We are going to demonstrate the usage of Support Vector Machine, neural networks, naïve Bayes classifier and random forests for this task. One of the goals is to compare the performance of these algorithms and decide which one performs better.

The structure of this document is the following. First, we are going to discuss the background material related to SVM, Naïve Bayes classifier, and artificial neural networks (ANN). Second, we are going to discuss the contents of the dataset and how we arrived to it (here we will discuss data capturing procedure, preprocessing and feature extraction). And finally, we are going to discuss the results of applying machine learning techniques.

This article is motivated by sergio42, and his article published on habr.com (<https://habr.com/ru/post/304926/>). The difference is that we use our own set of features and test broader range of the machine learning algorithms.

The source code of our effort can be found here: <https://github.com/dmitriykuptsov/traffic_analysis_with_ml>

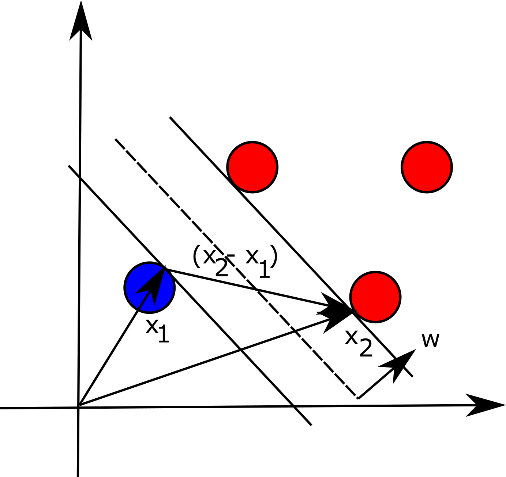
**Algorithms**

The number of machine learning algorithms which can be used for classification is rife: logistic regression, k-nearest neighbors, SVM, neural networks, naïve Bayes classifier, decision trees and random forests are just few. And in this article, we are going to demonstrate the results for SVM, artificial neural networks, naïve Bayes classifier and random forests.

**Support Vector Machine (SVM)**

Support vector machine is a powerful tool, and we believe that every data scientist needs to know the internals of this machine learning algorithm. And we are going to discuss the idea behind this classification algorithm in the proceeding paragraphs.

Given a hyperplane **wx**+b=0we can define two additional hyperplanes that are parallel to and equidistant from the given one. Namely, **wx**+b=1and **wx**+b=-1. If we say that we have a set of points (xi, yi)where yi is either one or negative one (depending on the class to which the sample belongs), we state that the following must hold yi(**wx**+b)-1**≥**0, with equality occurring when points are exactly on the hyperplanes. We visualize the problem for the 2-D setting in the figure below:



Now if we take two datapoints, one on each hyperplane, parallel to and equidistant to **wx+b=0**, **x1** and **x2,** the margin between these two points can be expressed as follows (basically, the difference between the two vectors projected onto unit vector **w**, which is perpendicular to hyperplane that separates the points in different classes):

According to SVM, the goal is to maximize this distance to reduce the classification error. However, maximizing this expression is the same as minimizing the following:

Since we are given constraints in a form yi(**wx**+b)-1**≥**0we can use Lagrange multipliers and construct the following equation:

To minimize this function, we need to find the partial derivatives with respect to vector **w** and b:

If we now plug in the values for **w** into the original equation and reduce it (here we also use the result for the second partial derivative), we obtain (dual optimization problem):

s.t.

We can now solve this maximization problem using Quadratic Programming (QP) solver and obtain Lagrange multipliers. Once this is done, we can find the vector **w** and bias term b. For the last point, we can take a point on the hyperplane:

The derivations that we have presented are good only for linearly separable samples, however, one can use so called kernel trick to classify more complex datasets. We leave out the discussion of the different kernels in here, but we will return to this issue when we will be playing with the empirical data.

**Naïve Bayes classifier**

Naïve Bayesian classifier is a simple approach to classify samples into multiple classes. Assume that we have vector **Y** of classes and vector of features **X**, we can then use the Bayes’ formula to find conditional probabilities as follows:

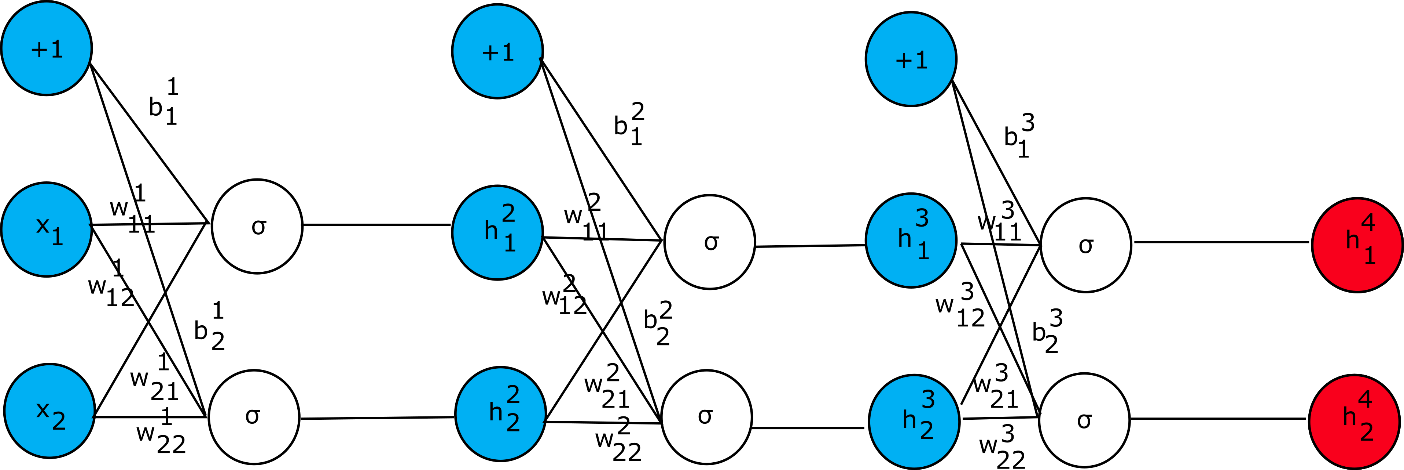
Calculating and is trivial: the first probability can be calculated as a fraction of time class appears in the dataset, and the second probability can be calculated as a fraction of time feature appears in the class . The above formula assumes that the features appear independently, hence the name of the classifier – naïve. The classification is therefore done as follows (we can ignore the denominator since it does not depend on class and is effectively a constant, i.e., it scales all results in a similar way):

To ease the computation, we can take the logarithm of the product to make it the sum.

**Artificial neural networks**

Neural networks are the powerful tools that are used when the task at hand is to classify the data.

Let’s consider simple artificial neural network that is shown below:



In this example, we consider that the error function is given as (we use mean square error (MSE) here for simplicity): , where k is the class number. In this case, the class labels should be encoded into array, in which 1 will be assigned to a true class and 0 will be given otherwise. The goal of the backpropagation is to adjust the weights of the neurons in such a manner that the error function is minimized. Let’s consider first how to adjust the weights of the neural network presented above and then we will give the general formula for the update rule.

In the derivations presented in the next few paragraphs we use sigmoid activation function . This function has a simple derivative which is convenient during the derivation of backpropagation rules.

Given the network presented in the figure above, let’s consider how different parts should be evaluated (starting from left to right):

Starting from the last layer (on the right) we can calculate the gradient with respect to weights as follows (for now let’s focus what effect does weight have on the error function):

We can denote:

Now in the similar fashion we can derive:

And

To succeed, we need to continue to derive partial derivatives for all the weights (including the bias terms) in the network. For example, computing the partial derivative with respect to and results in the following expressions:

We can observe the pattern in these expressions and so we can generalize. Let’s use the term such that:

Observing that we have recurrency we can restate the above equation as follows:

But, given that by the definition is:

We can write as

And therefore, we get:

With these equations at hand, we can write the partial derivative as follows:

We now have everything we need to construct the algorithm that updates the weights of the artificial neural network. Basically, a rather trivial approach is to evaluate step by step the artificial neural network (feedforward step) and obtain the outputs. Next compute the gradients and update the weights. Repeat these steps until the difference in errors is below preconfigured threshold (in other words, continue the update procedure until the error function does not change significantly). The weight update rule looks as follows:

Were is a learning rate, which is rather small value such as 0.01. It is the hyperparameter.

When working with neural networks, though, several other hyperparameters should be selected such as the number of hidden layers, the size of the hidden layers, the activation function, the optimizer (such as gradient descent or Adam optimizer), the learning rate parameter (which we described already) and regularization parameter. In our experiments we are going to use the feedforward neural network which consists of one input layer, two hidden layers and one output layer with k neurons. For simplicity, the activation function that we are going to use in the example is the sigmoid function. And finally, we are going to use the gradient descent algorithm for the optimization.

**Dataset**

Obviously, no network provider will allow us to capture the packets in their network for security and privacy reasons. Thus, we have sat down and captured the packets on a laptop for several hours. The original PCAP files were roughly 800MB and 1.3GB in size.

Before preprocessing the PCAP file contained the flows for the following applications:

|  |  |
| --- | --- |
| **Protocol** | **Number of flows** |
| TLS | 2550 |
| QUIC | 854 |
| Unknown | 713 |
| SSDP | 385 |
| DNS | 374 |
| IGMP | 287 |
| HTTP | 164 |
| MDNS | 79 |
| NetBIOS | 55 |
| LLMNR | 62 |
| WSD | 38 |
| ICMP | 30 |
| FTP (control) | 15 |
| DHCP | 14 |
| SSH | 13 |
| NTP | 8 |
| ICMPv6 | 6 |
| FTP (data) | 3 |
| AmazonAWS | 1 |
| SOCKS | 1 |
| OpenVPN | 1 |
| LotusNotes | 1 |

**Features extraction and preprocessing**

First step in preprocessing was to exclude all the flows, but those that run on top of TCP and UDP. We have also excluded the flows for which there were only few flows (we set the threshold to 50 flows). Next, we have ran the script to extract the features.

The process was the following: (i) for each TCP flow we excluded the handshake packets; (ii) we have

**Scores**

We use several metrics in our empirical evaluation.

**Results**

We have used Python and scikit-learn[[1]](#footnote-1) library in our experiments.

The other conclusion is that careful preprocessing of data and selection of features makes more than half way to successful classification of samples.

1. <https://scikit-learn.org/stable/> [↑](#footnote-ref-1)