Content based mime detection with byte frequency histograms

Project source repository

<https://github.com/LukeLiush/filetypeDetection>

JIRA issue with the TIKA feature

<https://issues.apache.org/jira/browse/TIKA-1582>

Tika has implemented 3 file type identification methods - magic bytes, glob and content metadata hint. The research is based on content based type identification, which extract and analyze the file byte histograms, this approach enhances identification and circumscribe it with what it has seen, so it only trusts the file with the type which has the similar byte histogram pattern it has seen, this has pros and cons, the pros is that it enhance the security aspect of the file type identification, but the cons is slow detection which requires the reading the entire bytes of a file for computing the byte histogram and it might be also myopic to the training data which might be less representative.

**Multi-class classification:**

Multi-class classification is challenging, because it is computational expensive and there are too many file types at least over 1000. The idea in this project is first to analyze the file types that Tika is failing to identify and minimize the problem domain, and target those unknown file types in this NASA polar scientific project and provide the support that help identify them.

As an example, GRB or GRB files are one of the unknown files, the content based learning research is targeting at the GRB file identification.

**Algorithms**

In this research, neural network and linear logistic regression are being implemented to classify GRB file type from other file types. Please note, one of the challenges is that if predicting non-GRB file types, it is better there are as many negative training examples as possible so as to train the model, however the size of negative training examples (i.e. non-GRB files) can be enormous, therefore it might be better we are given a set of types to be classified and we build a model that can work on those file type classification.

The input is byte histogram of a file and the output is a binary decision that predict whether the file is a GRB type or not a GRB type (note again, if we are given a set of types to be classified, the problem will be much simplified and the algorithm efficiency will probably be improved as the problem domain is narrowed and bounded).

The positive training examples are collected from the AMD polar web sites (\*.gsfc.nasa.gov). i.e. ftp://hydro1.sci.gsfc.nasa.gov/data/

The negative training examples are collected from the following i.e. <http://digitalcorpora.org/corp/files/govdocs1/zipfiles/>

**Preprocessing**

1. Read byte content of the file build byte histogram.

Build frequency by dividing each bin value with the max count of occurrence to have each bin value to fall in the range between 0 and 1.

1. Square the root to enhance the frequency distribution; as in some files some bytes have higher frequencies whereas other bytes are less frequent, or in a critical situation, some files have only one or two bins that occupy the majority of the count, this makes a large gap between the most frequent and less frequent, the solution is to apply a companding function - A law or u law; square-rooting the bin values also provide the same effect, so by considering the computational cost, the square the bin value is chosen in place of A law or u law.



A-law companding function curve



Square-root function curve

The following shows the difference

Byte frequencies **without** any companding.



Byte frequencies with A-law



The following is the a-alaw formula implementation in R.

alaw <- function(x, A=87.7){

 th = 1/A

 cond1 <- (x>=0 && x < th)

 cond2 <- (x>=th && x <= 1)

 x[cond1] <- A \* abs(x[cond1]) / (1+log(A))

 x[cond2] <- sign(x[cond2])\*(1+log(A\*abs(x[cond2])))/(1+log(A))

 x

}

The parameter x is the vector of frequency histogram.

The returned x is the vector of histogram after A-law is used.

For details of A-Law, please refer to <http://en.wikipedia.org/wiki/A-law_algorithm>.

Byte frequencies with square root (power of 1/2)



Byte frequencies with power of 1/3



As can be seen, if the power value becomes larger, the details behind that infrequent bytes will be a bit enhanced.

Because A-law requires a bit more computation overhead, the square-root is used as the companding function to enhance the details for those infrequent bytes.

**Data preparation**

Once GRB and non-GRB files are collected, the next step is to prepare our data set so as to allow R to easily manipulate.

We need to split the dataset into 3 chunks, training set, validation set and test set.

The dimensionality for each set is as follows.

m\*(256+1)

, where m indicates the number of training/validation/test examples; 256 is the size of features (i.e. byte frequency histogram with is **not** preprocessed with a companding function) + 1 for the labeled output.

All of the sets are treated as matrices which need to be saved as files; those files are loaded into the R program thru the ‘loadAndProcess.R’;

**Algorithm Neural networks**

The simple idea is that machine learning techniques such as neural network is used to classify the file types based on byte frequency histogram.

Neural network can be seen as a function, in this case its input is a vector of the preprocessed histogram and its output simply is a yes/no (1 or 0); With neural network, we can actually have a probability that might tell how likely it believes a given input histogram is a GRB or non-GRB, again it is worth stressing that non-GRB is a huge class to be classified, we might need to have a s many negative training examples as possible, but if we know what types we are dealing with, the problem might be further simplified with smaller set of classes;

**How to run the R program** with neural network based file content detection.

Required libraries:

 optimx

The entry or the main class is “main.R”

Go into R console or command line prompt, run the following

>source(‘main.R’)

This command will start the training and then outputting the model to a file called “tika.model”

The following screenshot shows logging information of the command.



**Gradient Checking**

In The main.R the gradientChecking utility has been commented out or disabled and gradient check is used to validate the gradient values produced by program and this generally helps to ensure whether the neural networks works as expected, sometime small error in gradients can still gives some good results but not optimal.

In order to enable gradientChecking, uncomment the following checkNNGradients shown as follows in the “main.R”



Please note, this is disabled by default, it is used when we want to validate the network, it is not recommended we enable gradientchecking when training our neural network model.

**Learning Curve**

Learning curve is also implemented to roughly help estimate how many training examples we need to build a good model; sometimes it is not necessary to train a model with a huge size of training examples, in order to find out how many examples we need for training, it is probably better to plot a learning curve where x-axis is the number of training examples and y is the corresponding error cost, if the error cost remain stable as more training examples are added, then it is probably time to cut the training data and use the portion of data from which the error starts to remain relatively small and unchanged.

In order to enable the learning curve, please uncomment the following highlighted line.



Please note, this is disabled by default, it is used when we want to find out the training examples, it is not recommended we enable it when training our neural network model.

**Linear logistic regression**

Linear logistic regression is also implemented initially to understand the data to be classified; it is simple and linear, but a bit more computationally inexpensive.

In order to enable it in main.R, please uncomment the following, but default only neural network is used for classification.



Linear logistic regression as its name says is linear and this linearity makes a bit limited and rigid in fitting into the training data; even though, we notice that GRB data collected from the AMD site potentially exhibit a simple linear pattern, still it is probably more safer to use a non-linear approach to classify GRB data, because our data are quite specific to AMD sites and we don’t want to force ourselves to believe that every other GRBs data all have a linear pattern; with neural network, a simple network structure is able to suffice for a better model, i.e. 2 hidden unit, it is not a too complex non-linear model but able to classify well.

So our test model is produced with 256 input units, 2 hidden units and 1 output unit.

The following shows a couple of preprocessed GRB file histograms from AMD site with Square-root.













Again it is worth stressing that if the number of types(say 5 types) to be classified is given or known in advance, then we can have 5 output units(i.e. 5 output neurons) that only concentrate on the 5 types, and this largely iron out the unknown i.e. the non-GRB files, because the challenge is the size of non-GRB data is enormously large and they cannot be clearly bounded or defined unless we have all types of files in the world in our training data, but there is even no global standard for every existing mime type in the world, and getting the entire files really does sound unrealistic; However, it does not mean we cannot build a negative decision region in our model, the idea is we harness our existing knowledge i.e. the existing data we have, we predict with it, even if the prediction is not 100% accurate, we can still build a model that can give a fair guess; Therefore if building the negative decision region (i.e. Non-GRB), the goal will be to have as much data as possible that gives more knowledge, and with more knowledge, our model will become more knowledgeable.

The following shows the some other types of file byte frequency histogram.

In contrast, the following shows some byte frequency histogram of other types.

**.doc**



008000.doc



008001.doc



008002.doc



008003.doc



008004.doc

**.xsl**



011010.xls



011005.xls



011006.xls



011008.xls



011009.xls

**.gz**



000579.gz (yohk\_softx\_fd\_19980813\_1625.fts)



000718.gz (000718 with type unknown)



000754.gz (000754.xml)



000736.gz (nsyebes.ps)



000786.gz (fermilab-pub-03-148-a.ps)

It is worth noting that the (compression gz) file histograms shown above seem to exhibit some sloppy and random patterns, it seems to have something to do with the files that are encapsulated inside the gz.

**.PS**



011090.ps



011460.ps



011824.ps



011879.ps



011884.ps

**.TXT**



001176.txt



001181.txt



001187.txt



001191.txt



001196.txt

**Output the model**

When finishing neural network training, in the end the model parameters and configuration (e.g. number of input units, hidden units, etc) are written in a text file called ‘tika-example.nnmodel’ in the same directory with ‘main.R’;

As we need to copy this file to Tika to allow Tika to detect the type for which the model is trained e.g. GRB type, note you can create many models for many different mime types, but GRB file type detection is discussed and used as one example to demonstrate the use.

The following line in main.R is the last line used to output the model, the name and structure can be customized according to different relish.



The exportNNParams method implementation resides in the utility class i.e. ‘myfunctions.R’; it can be also customized or replaced to create your own model file with different syntax or structure.

The following shows what the outputted model look like in that model text file.

The first line begins with # which indicates that this line is a model description that tells the type to be classified, the number of inputs, number of hidden units, output units and test set error cost; they are delimited by a tab.

The next line without # at the front shows a series of floating numbers separated by a tab, and they are model parameters, later we need to import the file into Tika and have the ExampleNNModelDetector to recreate the trained model with them in Tika so it can predict and classify the unseen file and determine with the imported model whether the given input file is a GRB or non-GRB type.



The following shows the printing formation produced by the R program after training in a bit more detail with the outputted/chosen model above.

[1] "Loading Dataset....."

[1] "Begining Training Neural Networks"

[1] "the length of weights 517"

[1] "The time taken for training: 330.257000"

[1] "The training error cost: 0.001380"

[1] "The validation error cost: 0.025099"

[1] "The testing error cost: 0.020883"

[1] "Training Accuracy: 100.000000"

[1] "Validation Accuracy: 99.650000"

[1] "Testing Accuracy: 99.762349"

**Tika Trained model Detector Design and usage:**

**Import the model into Tika**

Once the training is done, there is a model file that is generated as mentioned above. The above model file only have one model, however you can have multiple models written in that file or you can have several model files according to your needs.

Copy the ‘tika-example.nnmodel’ to the default directory of tika\tika-core\test \resources\org\apache\tika\detect\, alternatively in your own version of TrainedModelDetector, you invoke getDefaultModel with a different model file location, the purpose of this method is to read the model files and load those models into memory as an object instance i.e. TrainedModel; If your model file(s) have a different syntax or format, you might need overwrite this method getDefaultModel to provide reading and loading implementation that respect your syntax;

It is also possible that your model might use different size of input of byte histograms, some might consider a different bin size with some heuristics specific to their own data, in that case, it is possible to overwrite the readByteFrequencies(final InputStream input) :: TrainedModelDetector by providing your own version of byte histograms, and you also need to ensure the model parameters are used and set to reflect the same size of input.

TrainedModelDetector implements the Detector interface, but it is abstract meaning we need to subclass it with our own version of TrainedModelDetector.

ExampleNNModelDetector is its subclass, the purpose of subclassing the TrainedModelDetector is to supply the implementation of the method of loadDefaultModels that reads and registers the models into the model map <MediaType, TrainedModel> in the TrainedModelDetector. Once the model map is populated with a set of mappings with keys and values, the detect method in the TrainedModelDetector will be able to use the loaded models to predict the mime types.

The job of the TrainedModelDetector is to convert the given input stream to byte frequency histogram and pass that as the input to the models that have been loaded or registered in the map.

There is also a TrainedModel(abstract) and its subclass NNTrainedModel.

The TrainedModel is an abstract class that represents an abstraction of a trained model; a model object must have a method of “predict” with input of byte histogram vector, it returns a probability of prediction.

The following lists all of the classes for this feature (tika\tika-core\src\main\java)

org.apache.tika.detect.TrainedModelDetector (abstract)

org.apache.tika.detect.ExampleNNModelDetector

org.apache.tika.detect.TrainedModel (abstract)

org.apache.tika.detect.NNTrainedModel

Example model file (tika\tika-core\src\main\resources)

org.apache.tika.detect.tika-example.nnmodel

Unit test (tika\tika-core\src\test\java)

 org.apache.tika.detect. MimeDetectionWithNNTest