ContentMimeDetection

Tika - Content based MIME type Detection

JIRA issue with the TIKA feature

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

Motivation

The feature of TIKA-1582 is an extension of TIKA MIME detection based on file contents, i.e. the file byte histograms, and this feature provides a solution that follows a standard data mining process that extracts the knowledge out of the data (bytes). The motivation of this feature is to provide users with an option where content-based detection approach can be used, the "contents" can be defined in several ways, they can be the entire file bytes, byte n-grams, byte histograms, etc. In this feature, the byte-histogram is used as an example.

Some files are very huge in size, building byte histograms for those files requires significant amount of time, but it is worth noting that with domain specific knowledge or the heuristics (e.g. there might be some crucial and critical regions in the file that could help with the detection.), we can further reduce the amount of effort required for knowledge discovery or mining particular patterns that we can use in the type detection.

Please also note, this content based mime detection does require users to have some knowledge with data mining and machine learning, and the choice of learning algorithms used in the pattern mining does not seem to matter, the knowledge to be mined is the classification, and there are many classification learning algorithms invented or reinvented, the question of which one is the best depends on a goal and data, each of the learning algorithms requires lots of effort with thorough performance testing and emperical analysis, and some data might be linear separable, some are not; and a or a set of goals is very important as it often is in the context of performance tuning; we can also think about it as a performance tuning problem where we need to have a set of goals in terms of the scalability, complexity, accuracy, etc. And in order to set our goals, we might first need to understand our data, e.g. do we have enough data or what features do i need to use, do we need to transform input; and all those design questions seem to matter the most and highly depend on the user-specific data and more importantly they largely affact the choice of the algorithms , therefore we want to leave the choice of algorithms to users based on their goals and data in their environment.

As an example, we have actually implemented two algorithms for classifying the GRB file type from non-GRB types, one is linear logistic regression (gradient descent) and the other is neural network (back-propagation). Again, the neural network with back-propagation is a bit more complex with training and slower too, whereas the logistic regression is far cheaper in terms of complexity; and with the collected GRB data in our tests, it turns out that logistic regression also gives a good result with high accuracy, and it is worthy noting that it is always better to circumscribe the mime types to be detected; the example model attempts to classify grb files from non-grb files, and one of the observed challenges is to identify the non-grb file types whose class can be enormously large, the best practice is again to circumscribe a set of types to be classified, and the domain specific knowledge come into the play for well-defining a set of types in the user specific environment.

This feature could also enhance identification safety, so it only trusts the files that have similar byte histogram patterns it has seen in its training set, this has pros and cons, one of the pros as mentioned is that it enhance the security aspect of the MIME 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 biased or less representative.

Methods

As mentioned, the content-based mime detection follows a standard data mining process:

Raw data - > feature selection and data cleaning -> pre-processing and transformation -> learning patterns(machine learning) -> knowledge evaluation -> the use of knowledge(prediction/classification) In TIKA.

(It is worth noting that the feature selection requires learning the application domain)

Also please note the model has to be ready before it can be used in Tika; by "ready", we mean the model has to pass the final knowledge evaluation test. As shall be seen shortly, as an example Tika is only implementing the prediction phase, so the model parameters need to be loaded and read into Tika for prediction or classification; The process of training can be lengthy and tedious, sometimes training might need to be converted to parallel/map-reduce operations when training data is too large to fit memory, again this depends on the user's goal.

The following will briefly walk you through how the feature and example is implemented in this data problem. Please also refer to the attached docx for further information with the implemenation in R.

Please also refer to the code repo for details of the implementation for training a model, the neural network and logistic regression learning are all implemented in R and the following briefly describes the pre-processing and learning implementation in R and how to load the model parameters trained from the R programs into the Tika for mime detection.

The training program can be created or written in any programming language, the R implemenation is posted as an example, Tika only needs to load the well-trained model parameters from the training program and be able to use them to make predictions. The job of the feature in Tika generally have 4 steps as follow, and also it is flexible that you can overwrite the detect method of the TrainedModelDetector to define your own selected features if you have different features defined when training.

1. read the input in bytes
2. convert it to the byte histogram
3. preprocess and transform the histogram
4. predict the decision.

Project source repository

https://github.com/LukeLiush/filetypeDetection

The goal in the example model is to be able to classify GRB file types from non-GRB types.

Data preparation

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/

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.

We convert the stream of bytes to the histogram with 255 bins each of which stores a count of occurances, [it is also flexible that you define your own input with smaller number of histogram bins or the selected bins based on the domain knowledge, you can also apply a feature selection algorithm such as SOM, PCA or LCA when the features space may be too huge (e.g. you might want to work with the entire bytes as input variables), and you can also transform the input variables with the custom function(or kernel with svm) for the model to have non-linear effect, there are also many other practical tricks to achieve training a good model, but most of them might require a bit understanding with the application domain (i.e. in this case, the file types to be classified); To begin with, we probably need to understand our goal and the data (domain if possible), usually we need to visualize the data and we start with some simple algorithms to explore the data and then decide whether a more complex algorithm or function is needed].

Our training data have the 255 features each of which corresponds to a byte, and each training example is labelled with an actual output indicating its class.

Pre processing

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.

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-root is chosen to enhance the histogram detail in place of A law or u law.

https://lh6.googleusercontent.com/Soeeu7bv02MOLRulV9mMKy3WTb2RXU1PafO47m1g2_i8ATiVpBkTcgCozMG9VIgDENa7MYU-DbXctIK4iIWRZAnsJEg_Ye49tTN0FqnRrxmUsOTo3Ap9vaAeI4m9XiEceIeaIC4|height="260px;",width="561px;"

A-law companding function curve

https://lh6.googleusercontent.com/Soeeu7bv02MOLRulV9mMKy3WTb2RXU1PafO47m1g2_i8ATiVpBkTcgCozMG9VIgDENa7MYU-DbXctIK4iIWRZAnsJEg_Ye49tTN0FqnRrxmUsOTo3Ap9vaAeI4m9XiEceIeaIC4|height="260px;",width="561px;"

Square-root function curve

The following shows the difference

https://lh6.googleusercontent.com/Soeeu7bv02MOLRulV9mMKy3WTb2RXU1PafO47m1g2_i8ATiVpBkTcgCozMG9VIgDENa7MYU-DbXctIK4iIWRZAnsJEg_Ye49tTN0FqnRrxmUsOTo3Ap9vaAeI4m9XiEceIeaIC4|height="260px;",width="561px;"

Byte frequencies **without** any companding.

https://lh6.googleusercontent.com/Soeeu7bv02MOLRulV9mMKy3WTb2RXU1PafO47m1g2_i8ATiVpBkTcgCozMG9VIgDENa7MYU-DbXctIK4iIWRZAnsJEg_Ye49tTN0FqnRrxmUsOTo3Ap9vaAeI4m9XiEceIeaIC4|height="260px;",width="561px;"

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.

https://lh6.googleusercontent.com/lMLfrOj-Esc8kHSAY_Ly2em1qJLPFFczaKn0jvWDT13-OJDn8JPmOu8pHO72jCpK6SDmskWhjONDPRFBPw6vM-wCOfRNDiLtBISJNawABNQGuABnR8hF2vA4cWMmFXDoRSwtMGI|height="251px;",width="541px;"

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

https://lh6.googleusercontent.com/lMLfrOj-Esc8kHSAY_Ly2em1qJLPFFczaKn0jvWDT13-OJDn8JPmOu8pHO72jCpK6SDmskWhjONDPRFBPw6vM-wCOfRNDiLtBISJNawABNQGuABnR8hF2vA4cWMmFXDoRSwtMGI|height="251px;",width="541px;"

Byte frequencies with power of 1/3

Please also note, in order to make training a model converge quickly, we might also want to scale features down to a range from -1 and 1.

Training:

Once we have preprocessed our inputs, i.e. byte histograms, we are then ready to train a model with a machine learning algorithm.

The 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 again if we know what types we are dealing with, the problem might be further simplified with smaller set of classes; Also it is worthy noting, training with too many negative examples can also produce an unpromising result, in an extreme cases where you might have 10 positive examples and 10 million negative examples, with that huge difference it is likely you might have a biased model towards the one that dump everything it has seen into the negative class, so the choice of training set might be important, there are some cross-validation methods that might help assuage this bias .e.g we can randomly pick some portion of negative training data, but again the thorough performance testing is needed with each of the models you have trained based on the different data and the training parameters (e.g. a different regularized term, different network structure, etc). In additions, the choice of the structure or the tuning parameters depend on how well the model fit the data, when it over-fits the training data, we might want to adjust the regularized terms or add more training data; but when it under-fits, we might also want to increase the complexity of the network structure, but again the choice of structure depends on the patterns hidden in the data.

Training a linear logistic regression model seems to be far less complex compared to the neural network , a linear logistic regression can be implemented with svm, gradient descent, etc. It is a globally optimal solution as long as the data is linearly separable; and it is cheap in terms of computational complexity which is traded for accuracy; Again the choice of this algorrithm depends on the data; In the tests with the collected GRB training data, the trained logistic regression model also seem to generalize well with reasonably high accuracy.

Evaluation:

Once we finish training, we need to score our model and decide whether the model meets our goal, so the knowledge Evaluation is also very significant in the process,this is where the prepared test set is used. Again, the details of performance evaluation such as recall, precision, ROC, etc are skipped, but the idea is to decide whether our model meets our goals.

Use of the knowledge

'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.

```
https://lh4.googleusercontent.com/9NhU8MSntrg9JRxV55sG89v5MkBM_ZzI9wo5SoYN3chzirIB_R97VImM4LUc6Cps1wJSfDlZCAE-OdCCj6OGBmeGHyKn8falen0APY1UY0B4xgCZ1EUEX3JVYcqxznNEQ2ygXpw|height="27px;",width="602px;"
```

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.

```
https://lh6.googleusercontent.com
/ZkRhFs9ON4ELXTtClE9s0frCEsC_i7ktsWkmGlm10ktOCpJMorMB_UZA2K4pp6LIc8AK0c2LKhgss7ZQkhTop4eh9BBDYn-kQlC17PB21VUdMYjtvpHbUjY51XyS2iOgxSYjUIo|height="43px;",width="602px;"
```

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"

'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) *

${renderedContent}

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