

# Enhanced Metadata Generator & Data Context Similarity Analyzer

Atul Joshi, Shreyas Aserkar

Oklahoma State University, Stillwater.

(atjoshi, shreyas.aserkar)@okstate.edu

## ABSTRACT

Data is a valuable resource. Proper use of data can help people make better predictions, analyses and decisions. No matter how much effort is put into collecting a good dataset, errors will inevitably creep into the data, making it necessary to clean the dataset. Data cleaning can be complicated time consuming and expensive but is a necessary step. The research paper used in implementation of this project uses data association and data repairing to clean the data and uses techniques like Data Association, Data Context Similarity Analysis etc. Data Context Similarity Analysis has been implemented as part of this project.

## INTRODUCTION

### Overview

Our goal for this project is to implement Data Context Similarity analyzer which finds out the similarity degree between two datasets. In order to find out the similarity degree we need to generate the metadata of a dataset that can be used as the input to the DCSA that finds out the similarity degree between two datasets. Enhanced Metadata Generator generates the metadata, this EMG is important because it can assist in identifying data items, types and uniqueness for all types of datasets. To generate the metadata EMG uses neural networks. The neural networks have been implemented in two ways: Single Model and Multiple Models. The reason behind this was to test the accuracy of the EMG keeping in mind the performance factor as well.

## Results

To test the implementation, we have used two datasets “free-zipcode-dataset” and “country-capitals”. The metadata generated by both of the ways gave almost similar accuracies but there were performance issues.

### Single Model EMG Implementation

In this implementation the accuracy came out to be 0.7691 for free-zipcode-dataset with 15 columns in the dataset. The time taken to generate the metadata was 49 seconds.

### Multiple Models EMG Implementation

In this implementation the accuracy of the prediction results was 0.7835 for the same free-zipcode-dataset with same number of columns (15). But the time taken in this implementation was 3 minutes and 11 seconds that is about 4 times slower than the single model implementation. The performance of the multiple models EMG implementation will only degrade with the increase in the size of the dataset.

Data context similarity analyzer used metadata generated by the EMG to identify the similarity between two datasets. The final value for the two datasets came out to be 0.32. This shows that the two datasets are not very similar to each other. On the other hand, when we passed two exactly same datasets to DSCA, the similarity degree came out to be 1. And that tells us that the datasets are similar.

## DESIGN OF SOLUTION

Below mentioned is the implementation design for EMG & DCSA.

### Data Preprocessing

The ultimate goal is to find out how similar two datasets are and to do that metadata of a dataset is very important that is generated by the EMG. EMG uses neural network to generate the metadata. Now the dataset can have mix type of datatypes for example "String", "float", "integer" etc. The neural network however only deal with integers. So it was necessary to convert all the string values to the numeric and then pass it to the neural network. In order to generate the numeric values for string types. We created createDictionary.py. The input to this file is the dataset. How it works is, it takes the dataset as input, iteratively look for every value in the dataset and if the value is a string assigns a numeric index to it. The numeric index increases with the number of string values. Later the String and its corresponding index is written to an output file "dictionary.txt". EMG uses this file to replace the string with its particular index and pass those values to the neural network. Below is the figure of dictionary.txt.

```
AD:::::1
AE:::::2
AF:::::3
AG:::::4
AI:::::5
AL:::::6
AM:::::7
AO:::::8
AQ:::::9
AR:::::10
AS:::::11
AT:::::12
AU:::::13
AW:::::14
AX:::::15
AZ:::::16
Abu Dhabi:::::17
Abuja:::::18
Accra:::::19
Adamstown:::::20
Addis Ababa:::::21
Afghanistan:::::22
Africa:::::23
Aland Islands:::::24
Albania:::::25
Algeria:::::26
Algiers:::::27
Alofi:::::28
American Samoa:::::29
Amman:::::30
```

### Enhanced Metadata Generator Design

Enhanced metadata generator extracts the header names from the input dataset and uses them as the expected output label for the neural network. We have implemented the neural network in two ways:

#### Single Model Implementation

In the single model implementation, we create the neural network in a conventional way, we divide the dataset in two parts, training data and test data. We use the training data to create a prediction model and save it in the file system. Once the model is trained, we use the test data to make predictions (predict the header names). Once the header names are predicted successfully. We go through each value of each column of the dataset iteratively and check whether it is unique or not and also we find out the data type for each column. Finally, we generate a key value pair, with key as the header name, and value as the combination of uniqueness and data type.

<HeaderName>,<dataType><::><Uniqueness>

Below are the metadata generated by the EMG for the datasets that we have used in this project.

```
{
  "LocationType": "String::Non Unique",
  "Location": "String::Non Unique",
  "Long": "Float::Non Unique",
  "Zaxis": "Float::Non Unique",
  "LocationText": "String::Non Unique",
  "ZipCodeType": "String::Non Unique",
  "Yaxis": "Float::Non Unique",
  "City": "String::Non Unique",
  "Country": "String::Non Unique",
  "Zipcode": "Integer::Non Unique",
  "WorldRegion": "String::Non Unique",
  "Xaxis": "Float::Non Unique",
  "RecordNumber": "Integer::Unique",
  "State": "String::Non Unique",
  "Lat": "Float::Non Unique"
}
```

```
{
  "CapitalLatitude": "Float::Non Unique",
  "CapitalName": "String::Non Unique",
  "CountryCode": "String::Unique",
  "ContinentName": "String::Non Unique",
  "CountryName": "String::Unique",
  "CapitalLongitude": "Float::Non Unique"
}
```

## Multiple Models Implementation

In this EMG implementation, instead of making one neural network model for a dataset, we created a single model for every column in every dataset. So if there were 10 columns in a dataset, we created 10 neural networks. Every time a model was created it was saved in a list. When the test data was used to predict the header names. The whole column was used as the test data for the model, and if the predicted accuracy was less than 0.7 we ignored the model and iteratively jumped to the next model and this way we kept ignoring all the models until the best fitted model was found.

We have developed the neural network using Keras library. Keras wraps efficient numerical computation libraries Theano and Tensorflow.

## Neural Network Implementation

Following are the steps to develop a simple neural network:

1. Load Data
2. Define Model
3. Compile Model
4. Fit Model
5. Evaluate Model

### 1. Load Data

It is a good idea to set the random number seed. This helps in running the same code again and again and get the same result. This is helpful when we need to demonstrate a result, compare different algorithms using the same source of randomness. We can set the random seed using below command.

```
numpy.random.seed(7)
```

Now, we can load the data. We can simply use `Numpy.loadtxt()` to load a dataset, the return type of this method is 2-d matrix.

```
dataset = numpy.loadtxt("pima-indians-diabetes.csv",  
delimiter=",")
```

### 2. Define Model

We are using Keras python library to implement neural network and models in

Keras are defined as a sequence of layers. The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the "input\_dim" argument and setting it to a number equal to number of input variables. In this implementation we have used fully connected network with 3 layers. To define a fully connected network, we need "Dense" class. We can specify the number of neurons in the layer as the first argument, the initialization method as the second argument as "init" and specify the activation function using the "activation" argument.

```
Dense(x, input_dim=y,  
activation='relu')
```

We have used rectifier "relu" for the first two layer and "sigmoid" for the third layer. In the context of artificial neural networks, the rectifier is an activation function defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x),$$

where  $x$  is the input to a neuron. In computational networks, the activation function of a node defines the output of that node given an input or set of inputs.

We are using RLUs for the first two layers because RLUs, compared to sigmoid function or similar activation functions, allow for faster and effective training of deep neural architectures on large and complex datasets. We have used sigmoid function in the third layer to introduce nonlinearity in the model. The reason behind this is that if all the layers are linear then the whole network is equivalent to single layer. So there is no use of adding the layers in between a neural network.

```
Dense(1, activation='sigmoid')
```

### 3. Compile Model

Now once the model is defined we can compile it. Compiling the model uses libraries such as Theano or Tensorflow. We have used Tensorflow in our implementation. When compiling, we should specify some additional properties

required when training the network. We should specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training. In this case, we have used logarithmic loss, which for a binary classification problem is defined in Keras as “binary\_crossentropy”. We have also use the efficient gradient descent algorithm “adam” for no other reason that it is an efficient default.

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

#### 4. Fit Model

The model till now is compiled and now ready for computation. Fitting the model basically means training the model. We can use fit() method to train our model.

```
model.fit(X, Y, epochs=150, batch_size=10)
```

Here “X” is the training data and “Y” is the list of output variables (output labels).

The training process runs for a small number of iterations and uses a small batch size. These two can be specified using “epochs” and “batch\_size” parameters of fit() function.

#### 5. Evaluate Model

Once the model is trained we can evaluate the performance of the network on the test data by using predict method of the model.

```
model.predict(testData)
```

predict method returns the prediction results and based on these results we can do further analysis.

### Data Context Similarity Analyzer

The role of DCSA is to find out the similarity between two datasets. For this different techniques are used such as linguistic matching, structure matching and constraint matching. We have implemented constraint matching and structure matching as part of this project.

### Structure Matching

Structure matching attempts to match data items with similar features such as header names, datatypes, uniqueness etc. To perform this the structure matching consumes the metadata generated by the EMG and compares them to generate a structure similarity degree. Structure matching similarity values lie between 0 and 1. If the value is greater than 0.8 we can say that the two datasets have similar structural features. How it works is, it takes metadata of the two datasets as input. At first comparison of the keys is done. If the two keys are similar that means the header names of two datasets are similar, then we move on to the next step. The similarity is calculated using SequenceMatcher python library. Next, we compare the datatype and uniqueness of the data items. We take a mean of all the values which gives us the final degree of similarity for structural matching.

Below is the result of structure matching for two different datasets used as part of this project. As we can see the result for two different datasets came out to be 0.25 and this tells us that the two datasets are not very similar structurally.



Python console  
Console 4/A

```
In [2]: runfile('C:/Users/Atul/Independent_Studies/StructureMatching.py',  
Structural Matching Value: 0.25706074342409413  
  
In [3]:
```

Another test run for structure matching with same datasets. From the result below we can see the datasets are similar structurally as we have used same dataset for testing in this case.



Python console  
Console 7/A

```
In [2]: runfile('C:/Users/Atul/Independent_Studies/StructureMatching.py',  
Structural Matching Value: 1  
  
In [3]:
```

## Constraint Matching


Constraint matching aims to find out the similarity of constraint conditions among data items such as length etc. Similarity degree of the datasets is computed based on the relevant pairs of the similar names. Constraint matching similarity values lie between 0 and 1. If the value is greater than 0.8 we can say that the two datasets have similar constraints and similar names. Constraint matching takes two datasets as input and compares each value of two datasets, the SequenceMatcher finds out the similarity ratio of each element (actual value) of the dataset and calculates the mean. Also, the length of each element is compared with respective element of the other dataset and the mean is calculated. We take the means of two means and generate the final degree of similarity between two datasets. Below are the results of the test run of constraint matching on the two datasets used.

The below result is for two different datasets, the final value 0.36 indicates that the datasets are not very similar to each other.



```
Python console
Console 3/A
In [2]: runfile('C:/Users/Atul/Independent_Studies/ConstraintMatching.py')
Constraint Matching Value: 0.36443160942547055
In [3]:
```

Another test result with two same datasets gave the final value as 1 which tells us that the two datasets are similar to each other.



```
Python console
Console 5/A
In [2]: runfile('C:/Users/Atul/Independent_Studies/ConstraintMatching.py')
Constraint Matching Value: 1.0
In [3]:
```

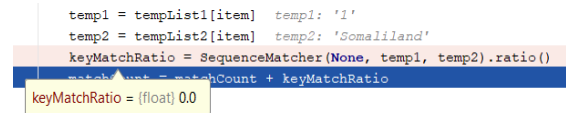
## SequenceMatcher

SequenceMatcher is an API provided by “difflib” library by python. It takes two string values as input and generates a similarity probability between two strings.

SequenceMatcher.ratio() returns a float in [0,1], measuring the similarity of the

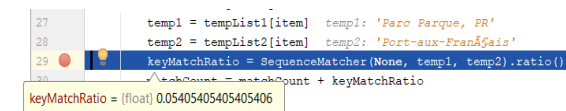
sequences. As a rule of thumb, a ratio value over 0.6 means the sequences are close matches.

In the below example we have used two strings “1” and “Somaliland”. The ratio came out to be 0.



```
temp1 = tempList1[item] temp1: '1'
temp2 = tempList2[item] temp2: 'Somaliland'
keyMatchRatio = SequenceMatcher(None, temp1, temp2).ratio()
keyMatchRatio = (float) 0.0
```

In another example the sample strings were “Parc Parque, PR” and “Port-aux-Fran” and the ratio came out to be 0.05.



```
27 temp1 = tempList1[item] temp1: 'Parc Parque, PR'
28 temp2 = tempList2[item] temp2: 'Port-aux-Fran'
29 keyMatchRatio = SequenceMatcher(None, temp1, temp2).ratio()
30 keyMatchRatio = (float) 0.05405405405405406
```

## Linguistic Matching

Linguistic matching attempts to match data items with linguistic features. The degree of similarity between datasets can be obtained from auxiliary information file. An auxiliary information file contains all relevant pairs having the same name. The similarity value lies between 0 and 1. When datasets have similar linguistic features the similarity measure will be as high as 0.8 to 1, otherwise the value will be less than 0.8. It is suggested that 0.8 will be a reasonable threshold value. We have not implemented linguistic matching as part of this project.

Combining the results of structure, constraint and linguistic matching we can identify that how similar are two datasets to each other.

## Conclusion and Future Work

From the above analysis, we saw that the results of the data context similarity analyzer depict the degree of similarity of two datasets. These results can be further used in the repairing of the dataset which will eventually

result in cleaning of the dataset. The current implementation has a good scope. The EMG results could include more parameters along with datatype and uniqueness. We have implemented `createDictionary.py` and generated `dictionary.txt` for the string values, instead of that we can use python libraries which can create numeric value for any string. This would save the efforts and time to write the code. We have implemented EMG in two ways single model and multiple models. In the single model when we train the model, once we train a model we can save it to a file system. But in the multiple models implementation we save the models in a list and not in a file system. We can save the list in cache or session or in cloud in some way so that we can update the models list with new models and can use the list to predict labels for new datasets. We can try and implement python assembler to enhance the performance of multiple models implementation of EMG. We tried that but we were unable to do so because of lack of resources. Implementing the linguistic matching will give additional accuracy to analysis of the DCSA.

## REFERENCES

- <https://docs.python.org/2/library/difflib.html>
- <https://pypi.python.org/pypi/editdistance>
- <http://federalgovernmentzipcodes.us/>
- <http://www.numpy.org/>
- <https://keras.io/>
- <https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/>
- <https://www.pyimagesearch.com/2016/11/14/installing-keras-with-tensorflow-backend/>
- <https://www.datacamp.com/community/tutorials/deep-learning-python>
- <https://docs.python.org/2.4/lib/sequencematcher-examples.html>
- <https://www.kaggle.com/>
- <https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7>
- <https://towardsdatascience.com/backward-propagation-for-feed-forward-networks-afdf9d038d21>
- Hong Liu, Ashwin Kumar TK, Johnson P Thomas, Xiaofei, "Cleaning Framework for BigData"
- Ashwin Kumar TK, Hong Liu, Johnson P Thomas, Goutam Mylavarapu, "Identifying Sensitive Data Items within Hadoop"