Enhanced Metadata Generator & Data Context Similarity Analyzer

Atul Joshi

Oklahoma State University, Stillwater.

atul.joshi@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.

KEYWORDS: spark, scala, enhanced metadata generator, data context similarity analyzer, structure matching, constraint matching, neural network

INTRODUCTION

Overview

The goal of 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 network model has been generated for every column of the dataset separately. It is called *Multiple Models EMG Implementation*. Once the metadata is generated, it is consumed by Data Context Similarity Analyzer to find out the similarity between two datasets. DCSA uses three types of matching to identify the similarity between the datasets:

1. Structure Matching
2. Constraint Matching
3. Linguistic Matching

As part of this implementation, only structure and constraint matching has been implemented. The implementation is done on *Spark version 2.0.1* framework and the language used is *Scala version 2.11.8*.

Results

To test the implementation, we have used two datasets *free-zipcode-database* and *country-capitals*. Below is table with column names and their type for the two datasets.

country-capitals		free-zipcode-database	
CountryCode	String	Country	String
CapitalLatitude	Double	ZipCodeType	String
CapitalLongitude	Double	City	String
CapitalName	String	State	String
CountryName	String	Location	String
ContinentName	String	WorldRegion	String
		Zipcode	Integer
		Lat	Double
		RecordNumber	Integer
		Long	Double
		LocationType	String
		Xaxis	Double
		LocationText	String
		Yaxis	Double
		Zaxis	Double

Multiple Models EMG Implementation

In this implementation the accuracy of the prediction results which was to identify the column names was 0.85 for the free-zipcode-dataset with 15 number of columns and 0.79 for country-capitals with 6 number of columns. As the implementation is done in spark, EMG takes very less amount of time to predict the column names.

Data context similarity analyzer uses metadata generated by the EMG to identify the similarity between two datasets. The final value for the two datasets came out to be 0.25. This shows that the two datasets are not very similar to each other. On the other hand, when I 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 is 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, *StringIndexer* of *org.apache.spark.ml.feature* has been used. We can change a column with String datatype to a numeric dataType using StringIndexer. Using StringIndexer we can give the column name of the column to be Indexed.

CountryName
Somaliland
South Georgia and...
French Southern a...
Palestine
Aland Islands
Nauru
Saint Martin
Tokelau
Western Sahara
Afghanistan
Albania
Algeria
American Samoa
Andorra
Angola
Anguilla
Antigua and Barbuda
Argentina
Armenia
Aruba

In the above screenshot, the column CountryName is String type and before feeding this to the neural network we need to convert the String values to numeric indexes. After using StringIndexer the final values are:

CountryName_INDEXER
13.0
1.0
221.0
142.0
47.0
203.0
210.0
109.0
159.0
40.0
123.0
21.0
115.0
19.0
25.0
64.0
94.0
72.0
232.0
228.0

This way we can convert all the columns with String data types to their respective numeric indexes. Also the dataset can have null/empty values, once the string indexing is done, all the null/empty values are replaced by 0.

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. The neural network has been implemented in multiple models fashion. One model for every column.

Multiple Models Implementation

Generally, the conventional way of implementing a neural network is to get the number of classes of the labels to be passed to the neural network, set the number of features, divide the dataset to training and test data. Train the neural network on the training data and then predict the results using the test data on the trained model. In this EMG implementation, instead of making one neural network model for a dataset, I have created a single model for every column in the dataset. So if there were 10 columns in a dataset, we created 10 neural networks. Every time a model is created it is saved in a list. The test data is used to predict the header names. The whole column is used as the test data for the model, and if the predicted accuracy is less than 0.7 the model is ignored and iteratively predictions are made on the next model and this way the models are ignored until the best fit model is found.I have developed the neural network using *org.apache.spark.ml.classification.MultilayerPerceptronClassifier* of spark ml lib.The output generated by the EMG is saved as *part* file in the below format:

<columnName>,<columnDatatype>,<uniqueness>

```
CountryCode,String,Nonunique
CapitalLatitude,Double,Nonunique
CapitalLongitude,Double,Nonunique
CapitalName,String,Nonunique
CountryName,String,Unique
ContinentName,String,Nonunique
```

Neural Network Implementation

Following are the steps to develop a simple neural network:

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

1. Load Data

   Once the data is cleaned and processed, we split the data to 70 and 30 percent to training and test data respectively. The training data is used to train the models (one for each column). Now once we have the test data, the next job is to extract every column iteratively and add a label column for that particular column. The *label* column is added so that it can be used as the output label for the neural network. The values of the *label* column will be the column name of the extracted column. For example, for the below column, the actual column name is *CountryName*.

   var finalClmnDataFrame = trainingData.withColumn("label", lit(trainingData.columns(i).dropRight(8)));

CountryName_INDEXER
13.0
1.0
221.0
142.0
47.0
203.0
210.0
109.0
159.0
40.0
123.0
21.0
115.0
19.0
25.0
64.0
94.0
72.0
232.0
228.0

   After adding the label column, the data frame is ready to be fed to the neural network, the updated data frame would look like below:

CountryName_INDEXER	label
13.0	CountryName
1.0	CountryName
221.0	CountryName
142.0	CountryName
47.0	CountryName
203.0	CountryName
210.0	CountryName
109.0	CountryName
159.0	CountryName
40.0	CountryName
123.0	CountryName
21.0	CountryName
115.0	CountryName
19.0	CountryName
25.0	CountryName
64.0	CountryName
94.0	CountryName
72.0	CountryName
232.0	CountryName
228.0	CountryName

Here, the *label* column depicts that every value has a output label of class CountryName. Once the *label* column is set, we again need to convert it to numeric type. So we again use StringIndexer on *label* column.

val label_indexer = new StringIndexer().setInputCol("label").setOutputCol("HEADER_INDEXER").fit(trainingData);

Next we set the number of features for this model, the features are input values for a particular class of output label which will be one as we only have one kind of values for a column.

val features = new VectorAssembler().setInputCols(droppedClmn).setOutputCol("features");

2. Define Model

Once the features are set for the model, we set the number of layers for the neural network, in this case the number of input layers will be one, as there is only one feature. The two intermediate layers are decided by hit and trial. Based on the accuracy of the predictions we decide the number of intermediate layers. The output layer will again be one as there is only one class of the output label.

val layers = Array[Int](1,4,5,1);

We have used org.apache.spark.ml.classification. MultilayerPerceptronClassifier to create neural network trainer. Multilayer perceptron classifier (MLPC) is a classifier based on the feed forward artificial neural network. MLPC consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the node's weights and bias and applying an activation function. Label converter is used to convert the numeric predicted results to their respective string values.

val droppedClmn = dataFrame.columns.slice(0,1);

val trainer = new MultilayerPerceptronClassifier().setLabelCol("HEADER_INDEXER").setFeaturesCol("features").setLayers(layers).setBlockSize(128).setSeed(1234L).setMaxIter(100);

val labelConverter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(label_indexer.labels);

3. Create Pipeline

ML Pipelines provide a uniform set of high-level APIs built on top of dataframe that help users create and tune practical machine learning pipelines. *MLlib* standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow. A *Pipeline* is specified as a sequence of *stages*, and each stage is either a *Transformer* or an *Estimator*. These stages are run in order, and the input DataFrame is transformed as it passes through each stage. For Transformer stages, the *transform()* method is called on the DataFrame. For Estimator stages, the *fit()* method is called to produce a Transformer(which becomes part of the PipelineModel, or fitted Pipeline), and

that Transformer's *transform()* method is called on the DataFrame.

In our case we have used *ML Pipelines* to configure a pipeline that has 4 stages:

1. label_indexer
2. features
3. trainer
4. labelConverter

val pipeline = new Pipeline().setStages(Array(label_indexer, features, trainer, labelConverter));

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.

   val model = pipeline.fit(trainingData);

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.

   val predictions = model.transform(lmnDataFrame);

   transform method return the datafarame with the predition results. To compute the accuracy of the model *org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator* has been used. *MulticlassClassificationEvaluator* is an Evaluator for multiclass classification, which expects two input columns: *prediction* and *label*.

   val evaluator = new MulticlassClassificationEvaluator().setLabelCol("HEADER_INDEXER").setPredictionCol("prediction").setMetricName("accuracy");

   Once the evaluator object is created, we can use *evalutate* method of evaluator which takes the *predictions* dataframe as input and generates a double value between 0 and 1.

   The predictions are made on all the models generated iteratively. Every time the accuracy of a model is evaluated, it is compared with the accuracy of the previous model and if the most recent accuracy is higher than the previous one. The most recent accuracy is updated. This way final accuracy of a particular column is calculated. The accuracies of all the columns are added together and then divided by the number of columns in the dataset. The result is the final accuracy of the EMG for that particular dataset.

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. The data context similarity analyzer uses the structural metadata to find similar datasets based on their names, data types and any other specific constraints. Scores of linguistic matching, structure matching and constraint matching are combined to identify similar datasets. All these three techniques are important because they provide an estimate of similarity based on names, structure and other user-defined constraints. Structure matching and constraint matching have been implemented 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. SequenceMatcher is a python library which is not implemented in spark scala. So I wrote the code by referring the algorithm from the 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 0.20 and this tells us that the two datasets are not very similar structurally.

```
degreeOfStructureMatching: Double = 0.1910961523715147
```

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.

```
degreeOfStructureMatching: Double = 1.0
```

Constraint Matching

Constraint matching aims to find out the similarity of constraint conditions among data items such as length, names 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. The below result is for two different datasets, the final value 0.30 indicates that the datasets are not very similar to each other.

```
degreeOfConstraintMatching: Double = 0.30445621505410847
```

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.

```
degreeOfConstraintMatching: Double = 1.0
```

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. Linguistic Matching has not been implemented 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.

Final Similarity Value

Once the similarity values of constraint matching and structure matching is computed. The final similarity value for two datasets, which tells us how similar two datasets are can be calculated by taking mean of the similarity values of constraint matching and structure matching.

```
degreeOfSimilarity: Double = 0.24777618371281157
```

The above results show that the two datasets are approximately 25% similar. The two datasets shown previously had some similar columns like country & countryName, CapitalLatitude & lat, CapitalLongitude & long and thus the results show that there is some similarity between two datasets. For exactly same datasets, the final value comes out to be

1. For the datasets are very similar the final similarity value should land in between 0.8 and 1.

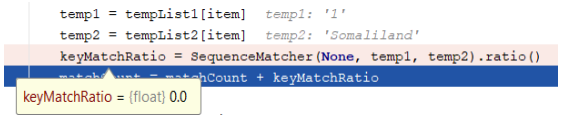
SequenceMatcher

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.

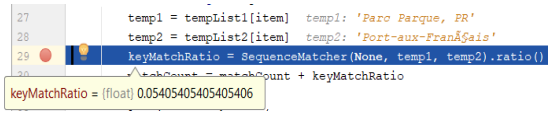
SequenceMatcher is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching". The basic idea is to find the longest contiguous matching subsequence that contains no "junk" elements (R-O doesn't address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people. SequenceMatcher tries to compute a "human-friendly diff" between two sequences.

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.



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



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 data type and uniqueness. We have implemented EMG with multiple neural network models. The data preprocessing could be done more efficiently; in this I have assigned 0 value to null/empty values. Based on different features of a class we can assign some expected value to empty/null values. When we train the model, 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 some kind of assembler to enhance the performance of multiple models implementation of EMG. I tried that but was 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://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
- https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier
- https://spark.apache.org/docs/1.5.2/ml-ann.html
- 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"