Metadata Extraction Using machine Learning (Stage 1)

Christopher Skorka 27/05/2019

**Important:** contains confidential information!

Table of Contents

[1. Introduction 2](#_Toc9984202)

[2. Abbreviations and Definitions 2](#_Toc9984203)

[3. Problem Definition and Algorithm 2](#_Toc9984204)

[3.1. Task Definition 2](#_Toc9984205)

[3.2. Algorithm Definition 4](#_Toc9984206)

[3.2.1. Process overview 4](#_Toc9984207)

[3.2.2. Data Representation 4](#_Toc9984208)

[3.2.3. Pre-processing 5](#_Toc9984209)

[3.2.4. Embedding 6](#_Toc9984210)

[3.2.4.1. Character Embedding 6](#_Toc9984211)

[3.2.4.2. Word Embedding 6](#_Toc9984212)

[3.2.5. Core model 7](#_Toc9984213)

[4. Experimental Evaluation 8](#_Toc9984214)

[4.1. Methodology 8](#_Toc9984215)

[4.2. Results 8](#_Toc9984216)

[5. Conclusion 9](#_Toc9984217)

# Introduction

The Geological Survey of Queensland (GSQ) collects and provides geoscience data, information and advice on Queensland’s mineral recourses and recourse potential through a range of geoscience projects and initiatives. GSQ needs to re-organize and restructure their data to allow for better access to it by both employees and customers. To achieve this we will move away from a relational database and to a graph based database. The existing relational databases are restricted to relationships that have been manually designed and are built into the data table structure. A graph bases database will allow for arbitrary links between data as well as running complex queries on data with no such restrictions found the traditional relational databases. In order to organize the existing data into this graph based system we need to generate rich and accurate metadata for existing files. This metadata is required to build the links between data objects. This is where this project comes in, the aim to solve this aspect of the system change. GSQ aims to develop machine learning software to extract and generate metadata from existing files by training, and improving a machine learned model that will be capable of processing files and the file structures that exists in to extract and generate useful labelled metadata. Our approach is to research existing similar problems and their solutions as well as develop machine learning software to test the viability of this technology. This stage of the project is concerned with verifying the viability of the application of the machine learning technology to this problem. At this stage it has been concluded that it can automate a very limited amount of metadata extraction however, further research can improve this.

# Abbreviations and Definitions

|  |  |
| --- | --- |
| ML | Machine learning |
| TF | Tensorflow |
| NN | Neural Network |
| RNN | Recurrent Neural Network |
| LSTM | Long-Short Term Memory |
| GRU | Gated Recurrent Units |
| Sequence | A variably sized sequence of objects (often temporal data) |
| Feature | One aspect of the data, ie metadata fields |

# Problem Definition and Algorithm

## Task Definition

3 main approaches for extracting data have been proposed thus far with the option of more approaches in the future.

1. File path metadata extraction
2. Document metadata extraction – extract data from file contents
3. Hierarchical metadata distribution – distribute metadata across relevant files based on their hierarchical position in the current file structure.

Stage 1 only address the first approach; file path metadata extraction. The goal is to develop a machine learned model that is able to extract at least some useful metadata about a file from its file path (complete file path not just the filename) and demonstrate the viability of using machine learning for this purpose. This stage does not concern itself with creating automated process that actually process live data and work into the new system.

The training data consists of a cvs file containing about 23900 samples. The file contains the following columns:

|  |  |  |  |
| --- | --- | --- | --- |
| Used | Column name | Data type | example |
|  | Unique Record ID | Numeric: positive integer | 132395 |
| Yes | FileName | String: file name only | GRAY\_83-NJX\_RAW\_MIGRATED\_QR007932\_132395.SGY |
| No | Original\_FileName | String: original file name | 83-NJX\_QR007932\_RAW\_MIGRATION.SGY |
| No | SurveyNum | Numeric: positive integer | 84022 |
| Yes | SurveyName | String | GRAY |
| Yes | LineName | String | 83-NJX |
| No | SurveyType | Categorical (1 type present) | LAND2D |
| No | PrimaryDataType | Categorical (1 type present) | PROCESSED\_SEGY |
| Yes | SecondaryDataType | Categorical (68 types) | RAW\_MIGRATED |
| No | TertiaryDataType | Categorical (7 types) | RAW\_MIGRATED |
| No | Quaternary | Categorical (2 types) | MIGRATED |
| No | File\_Range | String: Numeric range | 5001-5064 |
| No | First\_SP\_CDP | String: Numeric range | 100 |
| No | Last\_SP\_CDP | String: Numeric range | 350 |
| No | CompletionYear | Numeric: year | 1984 |
| No | TenureType | Categorical (4 types present) | EPP |
| Yes | Operator Name | String | DELHI PETROLEUM PTY LTD |
| No | GSQBarcode | String | QR007932 |
| No | EnergySource | Categorical (7 types present) | DYNAMITE SOURCE |
| Yes | LookupDOSFilePath | String: new complete file path | \SHUP\2D\_Surveys\Processed\_And\_Support\_Data\1980\GRAY\SEGY\GRAY\_83-NJX\_RAW\_MIGRATED\_QR007932\_132395.SGY |
| Yes | Source Of Data | Categorical (2 types) | GSQ |

Some data is not used at this stage either because it is not relevant or it usually won’t be found in the file path.

One example of relevant data that can be obtained from the file path is LineName. In this case the model will need to learn extract for example ‘80H-60’ from the file path ‘\SHUP\2D\_Surveys\Processed\_And\_Support\_Data\1980\KINNOUL\_BONNIE\_DOON\SEGY\KINNOUL\_BONNIE\_DOON\_**80H-60**\_STACK\_SDU10912TA\_129204.sgy’.

The type of processing is formally categorized as a sequence to sequence, meaning the input and output is a variably sized sequence of objects (characters). For this problem recurrent neural networks are best suited.

## Algorithm Definition

### Process overview

The process of can be broken down into 3 distinct stages: pre-processing the data, embedding and the core model.

The pre-processing is responsible with loading in the training data and processing it into a machine learning compatible format. This includes converting the textual data to vector representations, shuffling the dataset, and splitting the set into various desired subsets. In this step each character is converted into a vector and the entire dimensionality of the data set is normalized. The resulting vectors need to be of equal dimensionality regardless of sequence length. This is important as NNs are only able to train from equally sized numerical vector data.

In the next stage the training data is embedding into a denser vector space. As mentioned above the textual data is converted into a vector format on a character basis. Embedding is a form of dimensionality reduction. This process learns a transformation of the original vector representation into dense vector space. The embedding process learns both the transformation to encode the vector into the dense space as well as the transformation to decode it again. Embedding is applied twice, first to transform characters into a dense vector space and then to transforms sequences of characters or ‘words’ into even denser vector representations. After the model has processed the embedded training data its result will be decoded back into the standard vector representation on a character level using the decoder learned in this step. Embedding the data significantly increases the models accuracy and training speed.

The final stage is the core model of the processing pipeline. It is given embedded training data in embedded form, performs processing and returns a result in embedded form. It is comprised of an encoder and a decoder. Because input is a sequence, the encoder is a RNN that uses a single LSTM cell to encode the sequences or fords into an internal vector representation. The LSTM cell receives one embedded ‘word’ at a time and updates its internal state after each step. The output of the LSTM cell is a fixed sized vector after every word has been processed. A dense layer then decodes this output into the embedded word vector representation. Note that the decoder produces a fixed size vector and not a sequence, the program then simply removes training padding tokens.

### Data Representation

During the steps of processing the data in transformed into a number of different representations.

The raw training data is provided in the form of a csv file with a column for each of the feature of the data as described in [Task Definition](#_Task_Definition). During research the ‘LookupDOSFilePath’ and ‘FileName’ columns are used for inputs, the former is very similar to the real input that this model might operate on, the latter is a shorter substring of the former that is useful in testing models with shorter training times. Each field in the csv file is a variable length string or empty.

After pre-processing the data is contained in a dictionary that maps column names onto multidimensional arrays of onehot encoded vector representations the characters:

|  |
| --- |
| {  ‘path’: ndarray(n\_samples, n\_character, n\_vector\_length)  ‘p\_words’ : ndarray(n\_samples, n\_words, n\_character, n\_vector\_length)  …  } |

### Pre-processing

Pre-processing is comprised of the following steps:.

1. Available characters and token are defined and define char to int mappings
2. Read raw data
3. Read data into a dictionary mapping column names onto lists of strings
4. For selected columns, split strings into ‘words’
5. Vectorize strings by replacing characters with corresponding integers
6. Read integers into bounding multidimensional array, padded with the padding token
7. Convert integers to onehot encodings
8. Split dataset into training and test sets
9. Shuffle data subsets

After step 3 the data looks as follow:

|  |
| --- |
| {  ‘path’: [  ‘DIR/FILE.EXT’,  …  ],  …  } |

After step 4 some of the column’s strings are split into a list of ‘words’ at punctuation marks and symbols and the punctuations or symbols become a word themselves as well. This is only applied to some columns where it is reasonable.

|  |
| --- |
| ‘DIR/FILE.EXT’ -> [ ‘DIR’, ‘/’, ‘FILE’, ‘.’, ‘EXT’ ]  string -> list(string) |

After step 5 strings have been converted into lists f integers:

|  |
| --- |
| ‘DIR’ -> [ 4, 9, 18 ]  String -> list(int) |

After step 7 each integer is converted into a fixed length vector that uses the onehot encoding. The length of this vector is the number of characters and tokens available.

|  |
| --- |
| 2 -> [ 0, 0, 1, …, 0, 0, 0 ]  int -> list(floats) |

The final column’s data would be a multidimensional array of floats and have one of the following shapes:

|  |
| --- |
| If not split into words:  (n\_samples, n\_character, n\_vector\_length)  If split into words:  (n\_samples, n\_words, n\_character, n\_vector\_length) |

The final dictionary will map column names onto multidimensional arrays:

|  |
| --- |
| {  ‘path’: ndarray(n\_samples, n\_character, n\_vector\_length)  ‘p\_words’ : ndarray(n\_samples, n\_words, n\_character, n\_vector\_length)  …  } |

### Embedding

As part of the processing of file paths 2 levels of embedding take place. Firstly the onehot vectors representing a single character are encoded (embedded) into a dense vector space. Secondly sequences of dense character vectors are encoded into dense word vectors. Encoding the data into much dense vector spaces allows the NN to handle the data much better and due to the smaller data sizes training is much faster.

The encoders and decoders are creates using auto-encoders (NN). This is performed once per feature (column).

**Activation functions:**

The following activation functions have been tried: softplus, softsign, relu, sigmoid, hard sigmoid, exponential, linear, and none. The configurations shown in the specific models below have shown to perform best.

**Loss function:**

Categorical cross entropy is the chosen loss function for the training of the encoders and decoders as the outputs are categorical onehot encodings. Mean square error was also tested but performed significantly worse.

**Optimizer:**

The Adam optimizer is used.

### Character Embedding

The NN model for character embedding is very straight forward, one dense layer to encode the data and one dense layer to decode the data:

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Type** | **Output** | **Activation** |
| (Input) |  | Onehot character vector |  |
| Encode characters | Dense | Dense vector | None |
| Decode characters | Dense | Onehot character vector | Sigmoid |

After training these layers are duplicated and used in the more complex models.

**Hidden layer size (latent variable dimensionality):**

For a character set of around 50 characters and tokens a hidden size of 10 is the smallest size that effectively and accurately encodes the information.

### Word Embedding

The word embedding learns a latent vector space transformation for arrays of embedded characters. The character encoders and decoders are copied from the first character embedding model. But the character encoder and decoder layers are still trainable. Therefore this step trains a new character encoder/decoders as well as the word encoders/decoders. Training the character and word encoder together has shown to perform significantly better than fixing the character embedding layers. Setting the character weights to the pertained character character’s model speeds up the training of this setp.

Model summary:

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Type** | **Output** | **Activation** |
| (Input) |  | Array of onehot character vectors |  |
| Encode characters | Dense | Array of dense character vectors | None |
| Concatenate characters | Reshape | Concatenated character vector |  |
| Encode words | Dense | Dense word vector | None |
| Decode words | Dense | Concatenated character vector | Sigmoid |
| Split into characters | Reshape | Array of dense character vectors |  |
| Decode characters | Dense | Array of onehot character vectors | Sigmoid |

**Hidden layer size (latent variable dimensionality):**

For a words with around 20 characters (encoded) a hidden size of 35 is the smallest size that effectively and accurately encodes the information.

### Core model

Overall the model performs as follow: the character encoder encodes each character individually, then characters of a word are concatenated and each word is encoded individually. The LSTM then encodes the sequence of word vectors into a vector representing the entire input. This is then decoded into the encoded words. Each word is decoded into encoded characters and each encoded character is decoded into the onehot character vector.

The weights for the character and word encoders and decoders are copied from the pre-trained word embedding’s.

The core is comprised of a LSTM cell and a dense network. The LSTM cells is given the array of encoded word vectors and processes them sequentially. After each word vector it updates its internal state and once all words have been processed it outputs its hidden state. The dense layer then decodes the hidden state into an array of word vectors. The hidden size should a few time larger than the embedded size of the output so that I can contain the information of the output as well as some information regarding the current state of processing.

The total model architecture including all levels of encoders and decoders is the following:

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Type** | **Output** | **Activation** |
| (Input) |  | Array of onehot character vectors |  |
| Encode characters | Dense | Array of dense character vectors | None |
| Concatenate characters | Reshape | Concatenated character vector |  |
| Encode words | Dense | Dense word vector | None |
| Encode sequence | LSTM cell | Latent vector | Sigmoid |
| Decode sequence | Dense | Dense word vector | Sigmoid |
| Decode words | Dense | Concatenated character vector | Sigmoid |
| Split into characters | Reshape | Array of dense character vectors |  |
| Decode characters | Dense | Array of onehot character vectors | Sigmoid |

**Exploding gradient problem**

During training of the model as well as embeddings it would sometimes happen that the loss function suddenly returns a NaN value. This breaks the training process and changes all weights to NaN and consequently practically erasing the model.

This is cause by the exploding gradient problem where one or more weights keep increasing rapidly until their value exceeds the upper limit and their value becomes NaN.

To combat this kernel constraints are enforced. A value between 5 and 15 usually prevents the gradient explosions with little to no impact on accuracy.

# Experimental Evaluation

## Methodology

The main performance measure for the complete network is a custom written function that returns either a 0 or 1 for each prediction, where 1 indicates the prediction to a sample in vectorised form (Not onehot encoding) exactly matches the ground truth. The final number displayed is the percentage of predictions that are entirely correct.

The model has been trained with input FileName and on three target outputs (separate model instances), LineName, SurveyName, and GSQBarcode. Training was run for 100 epochs and training accuracies were returned for each epoch. After each test (100 epochs on a model instance) the accuracy was measured on an unseen test dataset. The results obtained from the final test are representative of performance towards new data.

## Results

|  |  |  |
| --- | --- | --- |
| **Target** | **Sample predictions** | **Testing accuracies** |
| LineName | “AR91\_**AR91-31**\_FINAL\_FILTERED\_SDU02007TA\_238981.SGY”  -> “**AR91-31**” | 91.5% Exact matches |
| SurveyName | “**NIELLA\_AND\_EXTENSION**\_85-N31\_FINAL\_FILTERED\_SDU02007TA\_238985.SGY”  -> “**NIELLA & EXTENSION**” | 93.0% Exact matches |
| GSQBarcode | “AR91\_AR91-31\_FINAL\_FILTERED\_**SDU02007TA**\_238981.SGY”  -> “**SDU02007TA**” | 94.2% Exact matches |

# Conclusion

Machine learning techniques can be applied to extract and generate rich and useful metadata from existing files. In this first stage, recurrent neural networks with LSTM cells combined with multiple levels of embedding have proven to be very effective in producing high accuracy models with testing results of above 90% on unseen test data. Therefore, machine learning techniques and specifically neural network technologies are an excellent approach to solve this task.