Column prediction using Recurrent Neural Networks

Sagar Khanna
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
sk6921@cs.rit.edu

***Abstract*— The recent text generation model using character embeddings is an efficient method for learning high-quality distributed vector representation that captures many precise syntactic and semantic character relationships. In this paper, I present an extension that can be applied to a distributed representation of a database column. Using known column names of a table, we train our model to generate new and meaningful column names.**

I. INTRODUCTION

Tables on the Web, often contain highly valuable data, are growing at an extremely high fast speed. These "data lakes" often lack metadata, which makes the structure of stored data challenging to understand. The language model has shown to be useful for improving natural language processing tasks. These include sentence-level tasks such as paraphrasing and inference, which aim to predict the relationship between tokens of a sentence.

There are two strategies for applying language representations: word level and character level. The character level approach, such as char2vec [2], uses symbolic embedding of words. Word level approach represents each sequence of a symbol of arbitrary length with a fixed vector and a distance metric that calculates the similarity. The word-level approach, like One-hot encoding, introduces a novel approach of creating an integer vector of each word in the vocabulary, and then the similarity is calculated by using a distance metric.

I argue that the current techniques restrict the power of representation, especially the character level approach. The major limitation is that the character level approach uses the context, i.e., the previously seen characters only to predict the new column names. Character level model does not consider the randomness of the column in the real world. In the context of any column in a table, i.e., other column names can be in any order.

In this paper, we improve the character-based approach by proposing Colum2Vec. It uses both character level approach and word-level approach. Our approach randomly picks a column name in a table and creates a word embedding for the current input space of the data. Column2Vec defines a relationship between how our target column name is related to other column names. The process of creating random target columns is called the Encoder step, where we create integer matrices of all the possible random combinations of a given set of column names. The Encoder step also removes the context dependency problem in character-level models as we choose each column randomly rather than in a specified order. We also show that the Colum2Vec approach helps us to generate meaningful column names as the Encoder step helps us to create relationships between each column, is used to predict the next character rather than just using the previously seen character. These encodings help the model to understand the semantic relationship between the characters and the words present in the column names.

Finally, the results show that the proposed model Column2Vec also generates very few non meaningful column names due to the additional word embedding used during training.12

II. BACKGROUND

Word2Vec [3], GloVe [11], and BERT [10] are the popular approaches for word embedding. Recently, other methods introduced that improve the performance of word embedding by using semantic information among the words. In Word2Vec, they find the words that appear more frequently and replace them with a unique token, i.e., It looks at the local frequency of words occurring together. Finally, the vector for all the phrases learned as a single word embedding. One of the features of this method is that all words and phrases are in the same vector space.
In GloVe, the words projected into a larger vector space where similar occurring words cluster together. But GloVe also considers the global statistics (word occurrence). Embedding of a sentence either at word level or character, the level has also found use in the domain outside Natural Language Processing (NLP), such as in graph/network representation [2,8], including entity resolution and concept modeling. Character level embedding model [1] using Recurrent Neural Networks are similar to the goal of this paper. We try to learn the semantic information of each column name in the database in a character-based model. One solution is using a character level encode, which encodes each character to a character embedding based on the previously embedded characters. The first step is converting the column names in a table into unique characters and then generate an embedding in a higher dimension for each character. Then, we train the model in which we update the embedding based on the sequence of characters that follow each other. In this work, for column names prediction we hypothesize the potential for discovering embedding for each column name that and then using aggregation operation to build an embedding for an entire table that is used to generate plausible names for the new column and show if there is any pattern associated with them

III. DATA COLLECTION AND CLEANING

Data for this project is collected from the web. The major issue with the data collected are as follows:

- **Inconsistency**: As wed data lakes consists of million of tables. There is a lot of inconsistency in naming convection used among them example, some tables consider space as a delimiter, some use '_' or other special character.

Data cleaning is performed after collection of the data. The steps used for data cleaning are as follows:

- **Removing unwanted characters**: The data consists of lot of special character like tabs, new lines so we run a cleaning process where we replace all this characters.
- **Using consistent tokens**: For this project I have decided to use '_' as a token inside column names and space as a token that differentiates two column tokens.

IV. CHARACTER MODEL

The architecture for our character model uses the Recurrent Neural Network, shown in Figure 1. It consists of a Recurrent Neural network where at each time step t, an RNN takes the input vector $Xt \in Rn$ and the hidden state $h_{t-1} \in R^m$ and produces the next hidden state h_t by applying the following recursive operation:

$$h_t = f(W_{xt} + U_h^{t-1} + b)$$

Here W, U, b are the parameters of the RNN. For each character in the corpus we encode it using dense one hot encoding. Dense encoding is used as it faster and feasible to run model on it. The model consists of the following layers:

- **Embedding layer**: This layer takes in batch size length of character encodings and generates an output vector of size 256.
- **Gated Recurrent Unit**: Gated recurrent units are a gating mechanism that takes care of when the hidden state must be updated or when it should be reset, i.e., If the RNN finds a symbol of great importance, it learns not to update the hidden state. Likewise, the RNN learns to skip irrelevant temporary importance.
- **Dense layer**: Dense layer is a regular deep connected layer which applies the following operation: This layer also converts the output to the length of vocab in our corpus so that it can be used in prediction.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (64, None, 256)           9848064
_________________________________________________________________
gru_1 (GRU)                  (64, None, 1024)          3938304
_________________________________________________________________
dense_1 (Dense)              (64, None, 38469)         39430725
=================================================================
Total params: 53,217,093
Trainable params: 53,217,093
Non-trainable params: 0
```

Figure 5. Model summary of the character model.

Output = activation (dot (input, kernel) + bias)

Working of the character model:

The following is a step wise description used to create a character model.

- **Data cleaning step**: As the data is collected from the web it has a lot of inconsistencies like similar column names are represented using different notations example: item id can be represented as item_id or item-id. Thus, data cleaning is preformed where the only allowed special tokens are ' ' and '_'. Some rows have extra spaces which have been stripped off for consistency.
- **Data generation**: In this step, the data is loaded into model using custom dataset generators which take in batch size as a parameter. This helps in progressive loading of the data set and not use high memory during training.
- **Data preparation**: This step includes creating one hot vector encoding of each character including '' and '_' as the model should learn when to produce special characters. To create the encoding vectors, we first sort the characters present in the corpus and then generate unique integers to each of the character.

```
tf.Tensor(
[[19  2 28 ... 18 32 19]
 [25 29  2 ... 34 19  2]
 [18 15 34 ... 19 18  2]
 ...
 [14 28 35 ... 29 26 14]
 [19 20 14 ... 23 34 19]
 [36 23 17 ... 23 34 39]], shape=(64, 100), dtype=int64) tf.Tensor(
[[ 2 28 29 ... 32 19 33]
 [29  2 15 ... 19  2 19]
 [15 34 19 ... 18  2 17]
 ...
 [28 35 26 ... 26 14 23]
 [20 14 17 ... 34 19 27]
 [23 17 19 ... 34 39  2]], shape=(64, 100), dtype=int64)
(64, 100, 41) # (batch_size, sequence_length, vocab_size)
```

Figure 4. Sample encoded vector for character model with batch size of 64. Sequence length of 100 and vocab size of the train data as 41.

- **Training RNN:** To train the model we pass four parameters to the model:
    1. **Vocab_Size:** This parameter is output dimension of the vector.
    2. **Embedding_dim:** This parameter is the Embedding layer dimension which maps the sequence of characters to a output vector.
    3. **Batch input:** The batch input is the corpus of characters we are training the model on.
    4. **Rnn_units:** This is the number of rnn units to be used for training the model.

```
Epoch 1/10
198/198 [==============================] - 8s 41ms/step - loss: 2.4686
Epoch 2/10
198/198 [==============================] - 8s 41ms/step - loss: 1.7209
Epoch 3/10
198/198 [==============================] - 8s 41ms/step - loss: 1.4245
Epoch 4/10
198/198 [==============================] - 8s 41ms/step - loss: 1.2690
Epoch 5/10
198/198 [==============================] - 8s 41ms/step - loss: 1.1629
Epoch 6/10
198/198 [==============================] - 8s 41ms/step - loss: 1.0785
Epoch 7/10
198/198 [==============================] - 8s 41ms/step - loss: 1.0085
Epoch 8/10
198/198 [==============================] - 8s 41ms/step - loss: 0.9459
Epoch 9/10
198/198 [==============================] - 8s 41ms/step - loss: 0.8886
Epoch 10/10
198/198 [==============================] - 8s 41ms/step - loss: 0.8378
```

Figure 7. Sample training of the character model for 10 epochs which shows the time taken and loss at each epoch.

- **Generation Phase:** For the generation phase we use two parameters: **Model** and **start_string.** The model parameter is the fully trained model which can be used to get the prediction for a given corpus. The **start_string** is a sequence of characters which is converted to a sequence of integer vectors and then generation is made based on several parameters like:
    1. **num_generate:** This parameter describes the number of characters to generate for the current sequence of input characters
    2. **Temperature:** This parameter lets the model know how random the oput should be more the temperature higher is the change of getting random data.

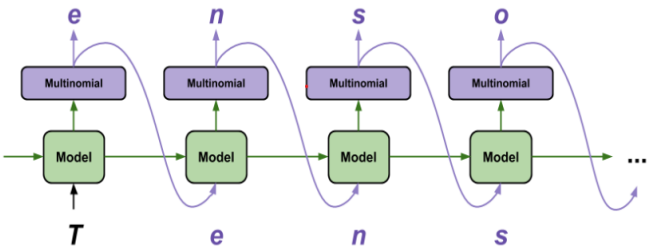

Figure 1. Character Model

V. COLUMN2VEC MODEL

The architecture of the character model uses Recurrent Neural Network, shown in Figure 2. It consists of a Recurrent Neural Network where each time step t, an RNN takes three inputs input vector Xt ∈ Rn , the hidden state $h_{t-1} \in R^m$ and M_t which is the embedding vector for each column token. It produces the next hidden state h_t by applying the following recursive operation:

$$h_t = f(W_{xt} + U_h^{t-1} + M_t + b)$$

Here Wht is character embedding of the present character and Uht-1 is previous hidden state and Mt is the column token embedding vector and b is the bias. The model for RNN consists of the following layers:

- **Embedding layer 1**: This layer takes input of the batch size which is the current character encoding.
- **GRU**: This layer is a Gated Recurrent Unit which takes care of when the hidden layer must be updated i.e., the RNN thinks the present character is important or to reset the model states.
- **Dense**: Dense layer is a regular deep connected layer which applies the following operation:

Output = activation (dot (input, kernel) + bias)

- **Embedding layer 2**: This embedding layer is an additional embedding vector we add to the RNN which keeps the column tokens encoding for the given batch size.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (64, None, 256)           10496
_________________________________________________________________
gru (GRU)                    (64, None, 1024)          3938304
_________________________________________________________________
dense (Dense)                (64, None, 41)            42025
_________________________________________________________________
embedding_1 (Embedding)      (64, None, 41, 256)       9848064
=================================================================
Total params: 13,838,889
Trainable params: 13,838,889
Non-trainable params: 0
```

The Column2Vec model uses two different types of encoding to train the model:

- Column name tokens encoding: The column names of every table are converted to one hot

vectors. These encodings are introduced to learn the relationship between column names during training. This helps the model to learn the pattern in the corpus.

- Character encoding: During the training phase each character are trained at each time step and the model is fed the character encoding.

Working of the Colum2Vec model:

The following is a step wise description used to create a character model.

- **Data cleaning step**: As the data is collected from the web it has a lot of inconsistencies like similar column names are represented using different notations example: item id can be represented as item_id or item-id. Thus, data cleaning is preformed where the only allowed special tokens are ‘’ and ‘_’. Some rows have extra spaces which have been stripped off for consistency.
- **Data generation**: In this step, I generate two type of dataset character level and column level. The character level dataset is same which we used for the character level model. The column level dataset is generated row wise for table.
- **Data preparation**: This step includes creating one hot vector encoding of each character including ‘’ and ‘_’ as the model should learn when to produce special characters. This step generates two type of encoding vectors: Firstly, character level and Secondly column token level. We then sort and store each of those created datasets and create one hot encoding vectors.

```
tf.Tensor(
[[28914 11667 21932 ... 21952 22679 18301]
 [11798 36397 37613 ... 20516  3563 20965]
 [26901 31483   990 ... 18301 24392 14545]
 ...
 [12296 12417 12419 ...  1222 27464  1322]
 [18301  1322 22188 ...  6541 35070 25226]
 [12614 22987 25109 ...  9406  9407 15605]], shape=(64, 100), dtype=int64) tf
[[11667 21932 22657 ... 22679 18301 18258]
 [36397 37613 20436 ...  3563 20965 25325]
 [31483   990 18301 ... 24392 14545 28753]
 ...
 [12417 12419 12150 ... 27464  1322  1430]
 [ 1322 22188 22193 ... 35070 25226 25227]
 [22987 25109 38304 ...  9407 15605 23606]], shape=(64, 100), dtype=int64)
(64, 100, 38469) # (batch_size, sequence_length, vocab_size)
```

Figure 5. Sample column level encoding of the training dataset with batch size of 64, sequence length of 100 and vocab size of 38649.

- **Training RNN:** To train the model we pass four parameters to the model:
    1. **Vocab_Size:** This parameter is output dimension of the vector.
    2. **Embedding_dim_1:** This parameter is the Embedding layer dimension which maps the sequence of characters to a output vector.
    3. **Batch input:** The batch input is the corpus of characters we are training the model on.
    4. **Rnn_units:** This is the number of rnn units to be used for training the model.
    5. **Embedding layer 2:** This layers takes in the column token level encoding which is used every time step.

```
Epoch 1/10
198/198 [==============================] - 462s 2s/step - loss: 3.1987
Epoch 2/10
198/198 [==============================] - 465s 2s/step - loss: 2.7442
Epoch 3/10
198/198 [==============================] - 466s 2s/step - loss: 2.6551
Epoch 4/10
198/198 [==============================] - 466s 2s/step - loss: 2.5958
Epoch 5/10
198/198 [==============================] - 465s 2s/step - loss: 2.5586
Epoch 6/10
198/198 [==============================] - 467s 2s/step - loss: 2.5330
Epoch 7/10
198/198 [==============================] - 461s 2s/step - loss: 2.5141
Epoch 8/10
198/198 [==============================] - 465s 2s/step - loss: 2.5023
Epoch 9/10
198/198 [==============================] - 460s 2s/step - loss: 2.4952
Epoch 10/10
198/198 [==============================] - 445s 2s/step - loss: 2.4900
Epoch 1/10
19/19 [==============================] - 42s 2s/step - loss: 18.2271
Epoch 2/10
19/19 [==============================] - 42s 2s/step - loss: 12.0415
Epoch 3/10
19/19 [==============================] - 42s 2s/step - loss: 11.9079
Epoch 4/10
19/19 [==============================] - 42s 2s/step - loss: 10.6055
Epoch 5/10
19/19 [==============================] - 42s 2s/step - loss: 10.3436
Epoch 6/10
19/19 [==============================] - 41s 2s/step - loss: 9.9456
Epoch 7/10
19/19 [==============================] - 41s 2s/step - loss: 9.7863
Epoch 8/10
19/19 [==============================] - 41s 2s/step - loss: 9.6061
Epoch 9/10
19/19 [==============================] - 41s 2s/step - loss: 9.4521
Epoch 10/10
19/19 [==============================] - 42s 2s/step - loss: 9.3275
```

Figure 6. Sample training of the model for 10 epochs which shows the time taken for each step and loss for each epoch.

- **Generation Phase:** For the generation phase we use two parameters: **Model** and **start_string.** The model parameter is the fully trained model which can be used to get the prediction for a given corpus. The **start_string** is a sequence of characters which is converted to a sequence of integer vectors and then generation is made based on several parameters like:
    1. **num_generate:** This parameter describes the number of characters to generate for the current sequence of input characters
    2. **Temperature:** This parameter lets the model know how random the output should be more the temperature higher is the change of getting random data.

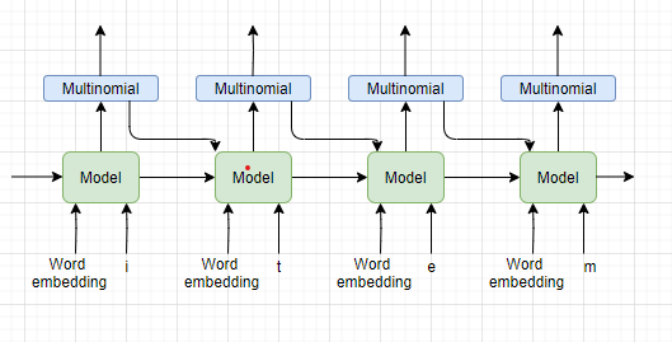

Figure 2. Column2Vec

VI. Results

Character Model:

The evaluation of each model is done by measuring meaningfulness of the predicted column names.

Randomness	Original input	Predicted column names
0.8	Country, zip code, rooms, accommodation	aid, tab_id, activity_id , description , inidc_agreement
0.7	Country, zip code, rooms, accommodation	production_count, other_subtitle, image_url image_width, created_at, updated_at
0.3	Country, zip code, rooms, accommodation	Rate, Summary, Buy, Folderid, Deleted, created_at, updated_at
0.2	Country, zip code, rooms, accommodation	created, modified, log_ip ,log_date ,log_time ,log_one_id ,created_at updated_at

The randomness of the model is how randomly it predicts the next character from the hidden state so that the model does not overfit.
The results from the column model shows that the model does not understand the relationship between the column tokens. Thus, the results as meaningful literals but not meaningful column names for the input columns.

Column2Vec Model

The evaluation of each model is done by measuring meaningfulness of the predicted column names.

Randomness	Original input	Predicted column names
0.8	Country, zip code, rooms, accommodation	checkin_date, handler type, city, status id name, description, username ,advert_id, name
0.7	Country, zip code, rooms, accommodation	checkin_date , id, status id ,created_at ,updated_at id ,action type ,external_id ,created
0.3	Country, zip code, rooms, accommodation	checkin_date ,date_created ,updated_at id ,username ,password , email
0.2	Country, zip code, rooms, accommodation	checkin_date, , realm_id, , station, sort_order activity, password id_admin, _shift_assign_log ,ticketa4 ,this_comments

The randomness of the model is how randomly it predicts the next character from the hidden state so that the model does not overfit.
The results from the column model shows that the model does not understand the relationship between the column tokens. Thus, the results as meaningful literals but not meaningful column names for the input columns.

ACKNOWLEDGMENT

I would like to express my sincere thanks to my capstone advisor Dr. Michael Mior and Dr. Alexander Ororbia, for guiding me throughout the project. Their timely feedback and providing access to the required resources for the project helped in the completion of this project. I would also like to thank Dr. Minseok Kwon for giving timely feedback on the posters and presentations. At last, I would like to thank the Computer Science department of Rochester Institute of Technology for giving me access to all the papers of elite publications for referring, which helped me understanding and completing the project. R

REFERENCES

[1] Zhang, Xiang, Junbo Zhao, and Yann LeCun. "Character-level convolutional networks for text classification." In Advances in neural information processing systems, pp. 649-657. 2015J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Chen, Jiaoyan, Ernesto JimÃĺnez-Ruiz, Ian Horrocks, and Charles Sutton. "Colnet: Embedding the semantics of web tables for column type prediction." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 29-36. 2019.

[3] Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 855-864. 2016.

[4] Hochreiter, Sepp, and JÃijrgen Schmidhuber. "Long short-term memory." Neural computation 9, no. 8 (1997): 1735-1780.

[5] Li, Shaohua, Tat-Seng Chua, Jun Zhu, and Chunyan Miao. "Generative topic embedding: a continuous representation of documents." In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 666-675. 2016.

[6] Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality." In Advances in neural information processing systems, pp. 3111-3119. 2013.

[7] Moody, Christopher E. "Mixing dirichlet topic models and word embeddings to make lda2vec." arXiv preprint arXiv:1605.02019 (2016).

[8] Narayanan, Annamalai, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. "graph2vec: Learning distributed representations of graphs." arXiv preprint arXiv:1707.05005 (2017).

[9] Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality." In Advances in neural information processing systems, pp. 3111-3119. 2013.

[10] Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pretraining of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

[11] Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation." In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pp. 1532-1543. 2014.