Language Modeling using SAS

JeeHyun Hwang[1], Haipeng Liu[2], Yang Xu[3]

[123]SAS Institute Inc., 100 SAS Campus Dr, Cary, NC 27513

Abstract
Language modeling is to predict next word in a given sentence. Language models are widely used for improving performance in many areas such as automatic speech recognition and machine translation. Prediction of next word is a sequential data prediction problem.
In this paper, we present two approaches for the task of language modeling. First, n-gram models are designed to statistically estimate the probability of next word given a sequence of previous words. This one is supported by SAS language model functionality, called language model action set. This action set is designed to efficiently train n-gram models on cloud platforms when a training data set consists of a large number of documents.
Second, we explore neural network for building language models using SAS deep learning functionality, called deep learning action set. This action set enables us to build LSTM-based models. We choose LSTM-based models because it is known that Long Short-Term Memory networks (LSTM) have advantages over recurrent neural networks in terms of handling exploding and vanishing gradient problems.
We conduct user studies and our user studies demonstrate the effectiveness of our language models.

Key Words: N-gram Models, Long Short-Term Memory network, Recurrent Neural Network, Deep Learning, Perplexity, Word Prediction

1. Introduction

Language models (LMs) [1, 2, 6, 7, 8] are used to predict next word in a given sentence. In other words, these models are designed to assign the probability of the last word of an n-gram given the previous words. These models are applied to a wide range of applications and domains with great success. Example applications include automatic speech recognition, machine translation, and spelling correction. The performance of these applications is greatly impacted by the quality of language models. For example, word error rate of automatic speech recognition is decreased by 18% by adopting language models [2] for experiments with Wall Street Journal data set.

Building language models of high performance is an important task. The performance of language models depends on several factors: methods for training, the amount of training data, and the quality of the training data. These factors are important to accommodate large amounts of training data into language models with high performance. Recently, large scale corpora that can be collected from the Web [1, 2] is used as training data.

In this paper, we focus on two approaches. First approach is n-gram models. N-grams represent a sequence of n words. N-gram models [6, 7, 8] are widely adopted methods by statistically estimating the probability of next word given a sequence of previous words. In particular, the size of the language model increases rapidly with growing number of n. In

consideration of the size, relatively low order, such as 1, 2, 3, or 4-grams are commonly used in practice. In addition, as these models are based on n-grams observed in the training data set, explicit probabilities of n-grams that are not observed in the training data set cannot be returned. In such cases, techniques such as stupid backoff [7] are used to yield better language models using empirically pre-computed probabilities with regards to n-1 grams.

Second approach is Long Short-Term Memory network (LSTM) [3] based models. LSTM is a family of recurrent neural networks. Recently recurrent neural networks have become increasingly popular for the task of language modeling. Similar to n-gram models, recurrent neural network language models estimate probability of next word. While n-gram models are based on n-grams observed in the training data set, this approach is based on the full history of text in the training data set. However, training recurrent neural networks using backpropagation is not trivial due to the well-known vanishing gradient problem [3]. The problem is that the gradient is not propagated properly and becomes to be vanishing or growing exponentially. To address this issue, Long Short-Term Memory (LSTM) [3] is proposed. In this network, the explicit gradient problem is prevented.

In this paper, we introduce how to build two types of language models using SAS.

- N-gram models are supported by SAS language model functionality, called language model action set. This action set enables users to build their own n-gram language models based on n-grams identified from input documents.

- LSTM models [3, 9, 10, 11] are supported by SAS deep learning functionality, called deep learning action set. We leverage LSTM models to predict next word in a sentence for language models

Moreover, our two approaches allow for processing a huge number of documents in a distributed computing resources on cloud. We conduct user studies and our user studies demonstrate the effectiveness of our language models.

2. N-Gram Model

This section introduces n-gram models for language modeling. We first describe approach, and then, details of each step to build n-gram language models. We next provide example code in Python.

2.1 Approach

We developed a language model action set that enables users to build their own n-gram language models based on n-grams identified from input documents. Note that action set is a collection of CAS actions. Each action performs analytics tasks on SAS Cloud Analytic Services (CAS) supported by SAS Viya platform. An action is designed to perform a task efficiently and effectively because this can work in parallel such as loading data and distributing the collection of n-grams statistics over several nodes in Cloud.

The language model action set, called langModel includes three actions to build n-gram language models in a pipeline: textToNGram, nGramCount, and nGramCountToLM actions. Figure 1 illustrates three actions. We describe these actions in Section 2.2.

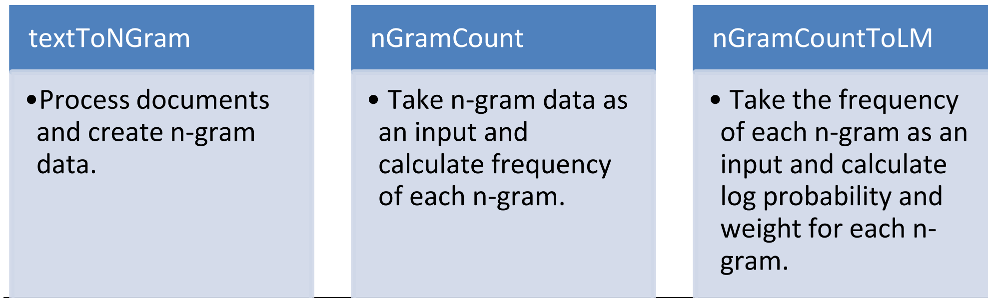

Figure 1. Overview of three actions that help build n-gram language models.

These three actions are dependent on each other in a pipeline. The first action takes input text as an input. Then, based on the output of the first action, the second action finds frequency of distinct n-gram. Then, third action, named nGramCountToLM analyzes frequency information and generates a language model where corresponding log probability and weight for each n-gram is stored. The language model is stored as binary SAS Cloud Analytics Service (CAS) tables. The model could be used for variety of purposes such as generating transcripts for speech-to-text applications. When this is used for the speech-to-text applications, the application's performance is improved in terms of accuracy.

2.2 Build N-Gram Language Models

This section presents each step. The first step is to use the textToNgram action, which converts raw text data into n-gram output table. N-gram output table contains 1-grams, 2-grams, …, and n-grams. An n-gram is a sequence of N words, w1, w2, w3, …, wn. Consider that an example sentence is "I like my car." For example, 2-gram (i.e., bigram) is a sequence of two words such as "I like", "like my", or "my car." 3-gram (i.e., trigram) is a sequence of three words such as "I like my", or "like my car."

The second step is to feed the n-gram table generated from the textToNgram action to the nGramCount action. Note that n-gram models are built based on the collection of n-grams and their frequency counts. In this step, the action identifies distinct n-gram and calculates frequency count of an n-gram.

The third step is to generate an n-gram language model using the nGramCountToLM action. The nGramCountToLm action takes frequency counts data for each n-gram and calculates probabilities and generates the language model. In this step, infrequent n-grams are pruned by setting a value for the minCount parameter within the action. This parameter specifies a minimum number of times that the n-gram must occur in the input data set. The action prunes any n-gram that does not appear as many times or more than this value. The language model output table includes probabilities and weights of n-grams:

- Probability represents conditional probability of the last word of an n-gram given the previous words. Let P (wn | w1, w2, w3, …, wn-1) is conditional probability for a sequence of N words.

  $$\mathrm{P}\,(\mathrm{wn} \mid \mathrm{w1}, \mathrm{w2}, \mathrm{w3}, \dots, \mathrm{wn}-1) = \ln\left(\frac{\mathrm{C}\,(\mathrm{w1}, \mathrm{w2}, \mathrm{w3}, \dots, \mathrm{wn})}{\mathrm{C}\,(\mathrm{w1}, \mathrm{w2}, \mathrm{w3}, \dots, \mathrm{wn}-1)}\right)$$

  where $\mathrm{C}\,(\mathrm{w1}, \mathrm{w2}, \mathrm{w3}, \dots, \mathrm{wn})$ denotes frequency counts of n-grams, w1, w2, w3, …, wn.

- Explicit probabilities of n-grams that are not observed in the training data set cannot be calculated. We use stupid backoff [7] to yield better language models using empirically pre-computed probabilities with regards to n-1 grams. We use 0.89 as a default value. Weights are calculated by 10 based log function on it.

2.3 Example Use Case

This example illustrates how to build n-gram language models. To perform tasks on CAS server, users submit code using CAS clients that are supported by several languages such as CASL [12], Python, Lua, or Java. Client-side source code in this example was written in Python.

```
Line #
     1   import swat
     2   s=swat.CAS("cloud.example.com", 5570)
     3   r=s.table.loadTable(path=" train.sashdat",
     4               casOut={"name":"train", "replace":True})
     5    s.loadactionset('langModel')
     6    s.textToNgram(table = {"name": "train"},
     7               casInVarList = ["Text"],
     8               nGrams = 4,
     9               casOut = {"name": "nGrams", "replace":True})
    10    s.nGramCount (table = {"name": "nGrams"},
    11               casOut = {"name": "nGramsCnt", "replace":True})
    12    s.nGramCountToLM (nGramTable = {"name": "nGramsCnt"},
    13                 minCount= 0,
    14                 casOut = {"name": "LMTable", "replace":True})
```

Figure 2: Code written in Python.

Figure 2 illustrates example code. In the example code, the SAS Scripting Wrapper for Analytics Transfer (SWAT) package [5] is used to interface with the CAS server at Line 1. Note that s is the session returned by SWAT. Next, train data set, named “train” is loaded into a CAS table at Lines 3-4. Table 1 presents all the data set in “train” data set. In the data set, “Text” column contained text data for processing.

Table 1: Example text table that is used as an input.

DocID	Text
1	I like my car
2	I like my house
3	I love my family

Next, langModel action set is loaded for using three actions for building language model at Line 5. To process “train” data set, three actions are called in a sequence:

textToNgram (Lines 6-9), nGramCount (Lines 10-11), and nGramCountToLM (Lines 12-14) actions.

Each action includes own parameters. For example, casInVarList and nGram parameters in the textToNGram action specify list of columns in the input table for processing and the maximum number of words that output table can contain. As we specify nGrams as 4 and casInVarList as "Text", output table, named "nGrams" includes only 1-4 grams collected from "train" data set in Figure 2.

Next, the nGramCountToLM action processes "nGrams" table and generates frequency count table, named "nGramCnt" at Lines 10-11. Finally, the nGramCountToLM action processes "nGramCnt" and generates an output table named "LMTable" that contains corresponding language model. In this step, minCount parameter specifies a minimum number of occurrences of a n-gram in the input table. To include all of n-grams, we set minCount as 0. Finally, Table 2 presents part of "LMTable." Table 2 shows an example output table after training a model. This table includes 1-4 grams and their corresponding log probability over terms and weight.

Table 2: Example output table of n-gram language model.

#	_NGRAM_1_	_NGRAM_2_	_NGRAM_3_	_NGRAM_4_	PROB	WEIGHT
1	I	love	my	family	0	NaN
2	I	like	my	car	-0.30103	NaN
3	I	like	my	house	-0.30103	NaN
4	love	my	family		0	-0.05061
5	like	my	car		-0.30103	-0.05061
6	like	my	house		-0.30103	-0.05061

* Actual column names of PROB and WEIGHT in the output table are _LOG_PROBABILITY_ and _LM_WEIGHT_.
* NaN represents a missing number.

3. LSTM-Based Model

This section introduces LSTM based models for language modeling. We first describe approach, and then, details of each step to build LSTM based language models. We next provide example code in Python.

3.1 Approach

LSTM models [3, 9, 10, 11] are supported by SAS deep learning functionality, called deep learning action set. We leverage the deep learning action set to predict next word in a sentence for language models. The deep learning action set, called deepLearn includes several actions to help build and train LSTM language models in a pipeline: buildModel, addLayer, and dlTrain actions. When the models are built, dlScore action is used to score an input table against the models. Figure 3 illustrates four actions. We describe these actions in Section 3.2.

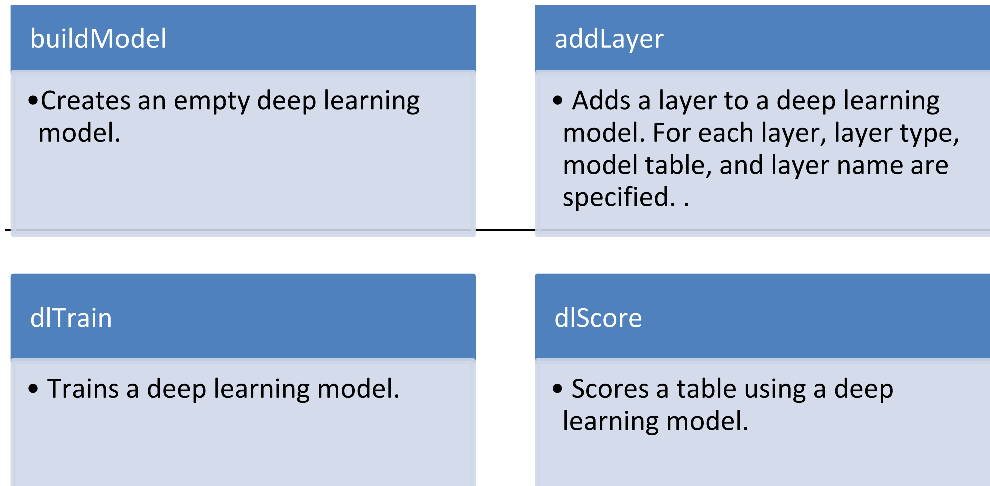

Figure 3. Overview of four actions that help build LSTM based language models

Figure 4 illustrates an overview of single LSTM layer. X(1), X(2), …, X(n) is a sequence of input words. As LSTM uses forget gate [3], LSTM splits each of cell state into two sequences. C(t) represents the current cell state at the time of t. H(t) is the current cell activation. The output y is the probability distribution of possible next word and this value is calculated by a sigmoid function based on the final cell state, H(3).

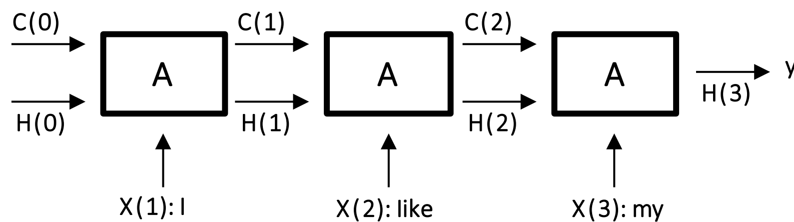

Figure 4: An overview of single LSTM layer

3.2 Build LSTM-Based Language Models

Figure 5 illustrates an example LSTM model. The model has an input layer X, two hidden layers s1 and s2, and output layer y. Input to the network in time t denotes as x(t), output is denoted as y. Input vector X(t) is formed by concatenating vector w representing word in X(t).

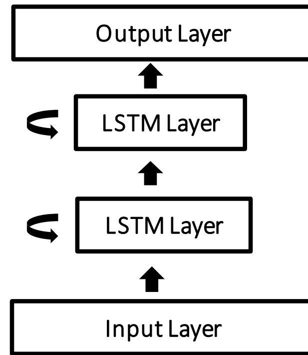

Figure 5: An example LSTM model.

When we build LSTM-based language models, we consider followings:

- At the input layer, the input words are mapped with corresponding vectors provided by a word embedding table.
- At the output layer, a softmax activation function is used to produce normalized probability values for candidate words.
- At the output layer, cross entropy loss function is used for training. This is equivalent to maximum likelihood.

3.3 Example Use Case

This example illustrates how to build LSTM-based models. Figure 6 illustrates example code. In the example code, s is the session returned by SWAT. Next, deepLearn action set is loaded for using actions for building LSTM language model at Line 1.

```
Line #
   1  s.loadactionset('deepLearn')
   2  s.buildmodel(model=dict(name='model', replace=True), type='RNN')
   3  s.addlayer(model='model', name='data', layer=dict(type='input'))
   4  s.addlayer(model='model', name='rnn1', srclayers=['data'],
   5       layer=dict(type='recurrent', init='XAVIER', n=100,
   6       rnnType='LSTM', outputType='samelength'))
   7  s.addlayer(model='model', name='rnn2', srclayers=['rnn1'],
   8       layer=dict(type='recurrent', init='XAVIER', n=100,
   9       rnnType='LSTM', act='TANH'))
  10  s.addlayer(model='model', name='outlayer', srclayers=['rnn2'],
  11       layer=dict(type='output', init='XAVIER',
  12       act='softmax', error='ENTROPY'))
```

Figure 6. LSTM models written in Python.

The buildModel action in the deep learning action set creates an empty “RNN” model at Line 2. The addLayer action adds an input layer (Line 3), two hidden LSTM layers

(Lines 4-9) with 100 neurons, and output layer (Line 12). For each layer, information such as layer type, model table, or layer name is specified. Details about each parameter are found in [4]. In particular, as noted in Section 3.2., at output layer, act='softmax' and error='ENTROPY' are specified for a softmax activation function and cross entropy error.

Next, Figure 7 illustrates example code to train an input model. Train and valid data sets, named “train” and “valid” are loaded into a CAS tables at Lines 1-4. These tables are created based on an output of the textToNgram action where 1, 2, 3, 4-grams are collected. We use only 4-grams in “train” table. Word embedding data set, named “glove100d” is loaded into a CAS table at Lines 5-6. The input words are mapped with corresponding vectors provided by the word embedding table. In particular, we set X(1), X(2), X(3) as an input variables and y as a target variable. The choice of the optimizer is important in the context of regularized models. In the example, we use Adam which is a variant of Stochastic gradient descent (SGD) with dropout as 0.5. We could use other deep learning parameters [4] that are supported in the deep learning action set.

```
Line #
 1  r=s.table.loadTable(path="train.sashdat",
 2          casOut={"name":"train ", "replace":True})
 3  r=s.table.loadTable(path="valid.sashdat",
 4          casOut={"name":"valid ", "replace":True})
 5  r=s.table.loadTable(path="glove100d.sashdat",
 6          casOut={"name":"glove100d", "replace":True})
 7  r = s.dlTrain(model='model',
 8    table=dict(name= 'train'), validTable=dict(name= 'valid'),
 9    inputs=["_NGRAM_1_", "_NGRAM_2_", "_NGRAM_3_"],
10    target='_NGRAM_4_',
11    texts=["_NGRAM_1_", "_NGRAM_2_", "_NGRAM_3_"],
12    nominals=["_NGRAM_4_"],
13    modelWeights=dict(name='model_weight', replace=True),
14    textParms=dict(initEmbeddings='glove100d',
15    modelOutputEmbeddings=dict(name='trainedGlove', replace=True)),
16    optimizer=dict(dropout= 0.5, algorithm=dict(method = 'ADAM',
17    learningrate = 0.005)),
```

Figure 7. Training code written in Python.

Next, Figure 8 illustrates example code to score an input test data set against the models. Test data set, named “test” is loaded into a CAS table at Lines 1-2. This table should be in the same format with “train” table. The dlscore action scores “test” data set and creates an output table, named “score”.

```
Line #
    1  r=s.table.loadTable(path="test.sashdat",
    2            casOut={"name":"test", "replace":True})
    1  r2 = s.dlscore(model='model',
    2     table='test',
    3     initEmbeddings='glove100d',
    4     initWeights= 'model_weight',
    5     copyVars=['_NGRAM_1_','_NGRAM_2_','_NGRAM_3_', '_NGRAM_4_'],
    6     casout=dict(name='score', replace=True))
```

Figure 8. Scoring code written in Python.

4. User Study

We have applied n-gram models on wiki data set. The wiki data set was used as a case study to explore how language model works. Table 3 lists training and testing data sets. # Obs column indicates the number of observations in the data set. Each observation represents a single sentence.

Table 3: Data set distribution.

Data Set	# Obs
Training dataset	86,892
Testing dataset	10,691

N-gram models are created based on tri-grams. Tri-grams uses only up to 3 sequential terms in the sentence. To perform this task, we used three actions, textToNGram, nGramCount, and nGramCountToLM actions. As generated language model can be extremely large, we reduce the number of rows by setting minCount parameter.

Table 4: Results measured for each training model

Model Name	minCount	# Obs	Perplexity
Arpar_model_0	0	1695056	125.74
Arpar_model_5	5	70843	201.93
Arpar_model_10	10	35159	216.86
Arpar_model_15	15	23195	222.17

Table 4 lists 4 language models that we created with minCount=0, 5, 10, and 15. # Obs column indicates the number of rows in the language model. Perplexity [2, 8] is measured against the testing data set. Lower perplexity shows better performance. From Table 4, we showed that, when minCount is 0, the perplexity is the lowest.

We trained wiki data set against LSTM based models that we built. The models were trained in several epochs [2, 4], in which all n-grams from the training data are sequentially used. After each epoch, we stored best models. We observed that perplexity continues to decrease after each epoch. We observed that perplexity is measured as 401.56 for the first epoch and dropped down as 278.11 within a few epochs. Approximately, this measure was reduced by more than 30%. However, the measure of the LSTM based models is still higher

than that of the n-gram models. We consider future improvements especially for LSTM based models. For example, to reduce this measure further, we plan to explore and train LSTM models using longer n-grams and sequence to sequence modeling. We also plan to investigate hyper-parameters of models and find the best configuration for these parameters to handle a large number of documents efficiently and effectively.

4. Conclusions and Future Work

In this paper, we presented two approaches that can be explored using SAS. We aim to predict next word in a sentence using CAS actions. First, we used SAS language model action set to build statistical n-gram language models. These models can be used to train and score a large number of documents efficiently. We also explored neural networks for building language models. Especially, we used LSTM-based models supported by SAS deep learning action set.

In Future, we would like to build language models with large-scale corpora. For this task, alternative training methods for LSTM models such as longer n-grams and sequence to sequence modeling need to be investigated for better performance. We also would like to apply our LSTM based models for various applications such as automatic speech recognition and machine translation. In addition, we would also like to investigate and find the best configuration for hyper-parameters of the models.

References

[1] Yoshua Bengio and Jean Sbastien Senecal. Quick training of probabilistic neural nets by importance sampling. In Proceedings of AISTATS, 2003.

[2] Tomas Mikolov et al. Recurrent neural network based language model. In Proceedings of INTERSPEECH, pages 1045–1048, 2010.

[3] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735– 1780, 1997.

[4] SAS Deep Learning Programming Guide https://support.sas.com/documentation/prod-p/vdmml/index_deep_learn_guide.html

[5] SAS Scripting Wrapper for Analytics Transfer (SWAT) https://github.com/sassoftware/python-swat

[6] Philip Clarkson et al. Statistical language modeling using the CMU-Cambridge toolkit. In Proceedings of EUROSPEECH, 2707-2710, 1997.

[7] Thorsten Brants and Ashok C. Popat and Peng Xu and Franz J. Och and Jeffrey Dean, Large language models in machine translation. In Proceedings of EMNLP, pages 858-867, 2007.

[8] Stolcke, Andreas, SRILM - an extensible language modeling toolkit, In Proceedings of ICSLP-2002, 901-904, 2002.

[9] Martin Sundermeyer, Ralf Schluter, and Hermann Ney. LSTM neural networks for language modeling. In Proceedings of INTERSPEECH, 194-197, 2012

[10] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research, vol. 3, pp. 115-143, 2003.

[11] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. Presented in NIPS Deep Learning and Representation Learning Workshop, 2014.

[12] SAS® Cloud Analytic Services 3.4: CASL Programmer's Guide