BlazingText algorithm

PDF | Kindle | RSS

The Amazon SageMaker BlazingText algorithm provides highly optimized implementations of the Word2vec and text classification algorithms. The Word2vec algorithm is useful for many downstream natural language processing (NLP) tasks, such as sentiment analysis, named entity recognition, machine translation, etc. Text classification is an important task for applications that perform web searches, information retrieval, ranking, and document classification.

The Word2vec algorithm maps words to high-quality distributed vectors. The resulting vector representation of a word is called a *word embedding*. Words that are semantically similar correspond to vectors that are close together. That way, word embeddings capture the semantic relationships between words.

Many natural language processing (NLP) applications learn word embeddings by training on large collections of documents. These pretrained vector representations provide information about semantics and word distributions that typically improves the generalizability of other models that are later trained on a more limited amount of data. Most implementations of the Word2vec algorithm are not optimized for multi-core CPU architectures. This makes it difficult to scale to large datasets.

With the BlazingText algorithm, you can scale to large datasets easily. Similar to Word2vec, it provides the Skip-gram and continuous bag-of-words (CBOW) training architectures. BlazingText's implementation of the supervised multi-class, multi-label text classification algorithm extends the fastText text classifier to use GPU acceleration with custom CUDA kernels. You can train a model on more than a billion words in a couple of minutes using a multi-core CPU or a GPU. And, you achieve performance on par with the state-of-the-art deep learning text classification algorithms.

The BlazingText algorithm is not parallelizable. For more information on parameters related to training, see Docker Registry Paths for SageMaker Built-in Algorithms.

The SageMaker BlazingText algorithms provides the following features:

- Accelerated training of the fastText text classifier on multi-core CPUs or a GPU and Word2Vec on GPUs using highly optimized CUDA kernels. For more information, see BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs.
- Enriched Word Vectors with Subword Information by learning vector representations for character n-grams. This approach enables BlazingText to generate meaningful vectors for out-of-vocabulary (OOV) words by representing their vectors as the sum of the character n-gram (subword) vectors.
- A `batch_skipgram` `mode` for the Word2Vec algorithm that allows faster training and distributed computation across multiple CPU nodes. The `batch_skipgram` `mode` does mini-batching using the Negative Sample Sharing strategy to convert level-1 BLAS operations into level-3 BLAS operations. This efficiently leverages the multiply-add instructions of modern architectures. For more information, see Parallelizing Word2Vec in Shared and Distributed Memory.

To summarize, the following modes are supported by BlazingText on different types instances:

Modes	Word2Vec (Unsupervised Learning)	Text Classification (Supervised Learning)

Single CPU instance	`cbow`
	`Skip-gram`
	`Batch Skip-gram`
		`supervised`

Single GPU instance (with 1 or more GPUs)	`cbow` `Skip-gram`	`supervised` with one GPU
Multiple CPU instances	`Batch Skip-gram`	None

For more information about the mathematics behind BlazingText, see BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs.

Topics

Input/Output Interface for the BlazingText Algorithm

The BlazingText algorithm expects a single preprocessed text file with space-separated tokens. Each line in the file should contain a single sentence. If you need to train on multiple text files, concatenate them into one file and upload the file in the respective channel.

Training and Validation Data Format

Training and Validation Data Format for the Word2Vec Algorithm

For Word2Vec training, upload the file under the *train* channel. No other channels are supported. The file should contain a training sentence per line.

Training and Validation Data Format for the Text Classification Algorithm

For supervised mode, you can train with file mode or with the augmented manifest text format.

Train with File Mode

For `supervised` mode, the training/validation file should contain a training sentence per line along with the labels. Labels are words that are prefixed by the string *__label__*. Here is an example of a training/validation file:

```
__label__4  linux ready for prime time , intel says , despite all the linux hype , the open-source

__label__2  bowled by the slower one again , kolkata , november 14 the past caught up with sourav
```

> **Note**
>
> The order of labels within the sentence doesn't matter.

Upload the training file under the train channel, and optionally upload the validation file under the validation channel.

Train with Augmented Manifest Text Format

The supervised mode also supports the augmented manifest format, which enables you to do training in pipe mode without needing to create RecordIO files. While using the format, an S3 manifest file needs to be generated that contains the list of sentences and their corresponding labels. The manifest file format should be in JSON Lines format in which each line represents one sample. The sentences are specified using the `source` tag and the label can be specified using the `label` tag. Both `source` and `label` tags should be provided under the `AttributeNames` parameter value as specified in the request.

```
{"source":"linux ready for prime time , intel says , despite all the linux hype", "label":1}
{"source":"bowled by the slower one again , kolkata , november 14 the past caught up with sourav
```

Multi-label training is also supported by specifying a JSON array of labels.

```
{"source":"linux ready for prime time , intel says , despite all the linux hype", "label": [1
{"source":"bowled by the slower one again , kolkata , november 14 the past caught up with sourav
```

For more information on augmented manifest files, see Provide Dataset Metadata to Training Jobs with an Augmented Manifest File.

Model Artifacts and Inference

Model Artifacts for the Word2Vec Algorithm

For Word2Vec training, the model artifacts consist of *vectors.txt*, which contains words-to-vectors mapping, and *vectors.bin*, a binary used by BlazingText for hosting, inference, or both. *vectors.txt* stores the vectors in a format that is compatible with other tools like Gensim and Spacy. For example, a Gensim user can run the following commands to load the vectors.txt file:

```
from gensim.models import KeyedVectors
word_vectors = KeyedVectors.load_word2vec_format('vectors.txt', binary=False)
word_vectors.most_similar(positive=['woman', 'king'], negative=['man'])
word_vectors.doesnt_match("breakfast cereal dinner lunch".split())
```

If the evaluation parameter is set to `True`, an additional file, *eval.json*, is created. This file contains the similarity evaluation results (using Spearman's rank correlation coefficients) on WS-353 dataset. The number of words from the WS-353 dataset that aren't there in the training corpus are reported.

For inference requests, the model accepts a JSON file containing a list of strings and returns a list of vectors. If the word is not found in vocabulary, inference returns a vector of zeros. If subwords is set to `True` during training, the model is able to generate vectors for out-of-vocabulary (OOV) words.

Sample JSON Request

Mime-type: `application/json`

```
{
"instances": ["word1", "word2", "word3"]
}
```

Model Artifacts for the Text Classification Algorithm

Training with supervised outputs creates a *model.bin* file that can be consumed by BlazingText hosting. For inference, the BlazingText model accepts a JSON file containing a list of sentences and returns a list of corresponding predicted labels and probability scores. Each sentence is expected to be a string with space-separated tokens, words, or both.

Sample JSON Request

Mime-type: `application/json`

```
{
 "instances": ["the movie was excellent", "i did not like the plot ."]
}
```

By default, the server returns only one prediction, the one with the highest probability. For retrieving the top *k* predictions, you can set *k* in the configuration, as follows:

```
{
 "instances": ["the movie was excellent", "i did not like the plot ."],
 "configuration": {"k": 2}
}
```

For BlazingText, the `content-type` and `accept` parameters must be equal. For batch transform, they both need to be `application/jsonlines`. If they differ, the `Accept` field is ignored. The format for input follows:

```
content-type: application/jsonlines

{"source": "source_0"}
{"source": "source_1"}

if you need to pass the value of k for top-k, then you can do it in the following way:

{"source": "source_0", "k": 2}
{"source": "source_1", "k": 3}
```

The format for output follows:

```
accept: application/jsonlines


{"prob": [prob_1], "label": ["__label__1"]}
{"prob": [prob_1], "label": ["__label__1"]}

If you have passed the value of k to be more than 1, then response will be in this format:

{"prob": [prob_1, prob_2], "label": ["__label__1", "__label__2"]}
{"prob": [prob_1, prob_2], "label": ["__label__1", "__label__2"]}
```

For both supervised (text classification) and unsupervised (Word2Vec) modes, the binaries (**.bin*) produced by BlazingText can be cross-consumed by fastText and vice versa. You can use binaries produced by BlazingText by fastText. Likewise, you can host the model binaries created with fastText using BlazingText.

Here is an example of how to use a model generated with BlazingText with fastText:

```
#Download the model artifact from S3
aws s3 cp s3://<YOUR_S3_BUCKET>/<PREFIX>/model.tar.gz model.tar.gz

#Unzip the model archive
tar -xzf model.tar.gz

#Use the model archive with fastText
fasttext predict ./model.bin test.txt
```

However, the binaries are only supported when training on CPU and single GPU; training on multi-GPU will not produce binaries.

For more details on dataset formats and model hosting, see the example notebooks Text Classification with the BlazingText Algorithm, FastText Models, and Generating Subword Embeddings with the Word2Vec Algorithm.

EC2 Instance Recommendation for the BlazingText Algorithm

For `cbow` and `skipgram` modes, BlazingText supports single CPU and single GPU instances. Both of these modes support learning of `subwords` embeddings. To achieve the highest speed without compromising accuracy, we recommend that you use an ml.p3.2xlarge instance.

For `batch_skipgram` mode, BlazingText supports single or multiple CPU instances. When training on multiple instances, set the value of the `S3DataDistributionType` field of the `S3DataSource` object that you pass to `CreateTrainingJob` to `FullyReplicated`. BlazingText takes care of distributing data across machines.

For the supervised text classification mode, a C5 instance is recommended if the training dataset is less than 2 GB. For larger datasets, use an instance with a single GPU (ml.p2.xlarge or ml.p3.2xlarge).

BlazingText Sample Notebooks

For a sample notebook that uses the SageMaker BlazingText algorithm to train and deploy supervised binary and multiclass classification models, see Blazing Text classification on the DBPedia dataset. For instructions for creating and accessing Jupyter notebook instances that you can use to run the example in SageMaker, see

Use Amazon SageMaker Notebook Instances. After creating and opening a notebook instance, choose the **SageMaker Examples** tab to see a list of all the SageMaker examples. The topic modeling example notebooks that use the Blazing Text are located in the **Introduction to Amazon algorithms** section. To open a notebook, choose its **Use** tab, then choose **Create copy**.

Did this page help you?

Yes No

Provide feedback

Edit this page on GitHub

Previous topic: Logs

Next topic: Hyperparameters

Need help?

- Try the forums
- Connect with an AWS IQ expert