Wordbatch - Technical Review

https://github.com/anttttti/Wordbatch

A parallel text feature extraction for machine learning.

Danilo Canivel
Hollywood - FL, United States
Email: canivel2@illinois.edu

Extract features from text has become a simple task with all this toolkits available, Scikit, Keras, NLTK, scapy, metapy and many others can do a very good job on this matter, but when you have to deal with huge amounts of text what happens? It can takes hours, days to finish and sometimes, even after hours waiting, you run out of memory or the machine freezes.

In general, what these libraries try to accomplish is to process all the data at the same time, producing very high dimensional data all at once, which for small datasets works great, but for the real world data, you are going to face processing and memory issues, if you do not use a solution to deal of the data in batches and/or multiple clusters.

In that way, Wordbatch was created to solve this processing issue for large amounts of data.

Wordbatch, as the name says, works with large mini batches of text data. As it process each batch it will internally store the statistics generate for it and will apply it on the task it is set to perform, like online IDF weighting for example.

With that said, you can imagine, as bigger the size of the batch, better will be the extraction, better will be the choices Wordbatch will made to extract meaningful features.

Currently Wordbatch has four feature extractors, WordHash, WordBag, WordVec, WordSeq.

The Extractors

WordHash

A wrapper around Scikit-Learn HashVectorizer, the extractor convert a collection of text documents to a matrix of token occurrences.
Using it from Wordbatch implementation you will have the ability to parallelize the extraction, increasing significantly the amount of time for processing for large amount of observations.

Let's make some tests:

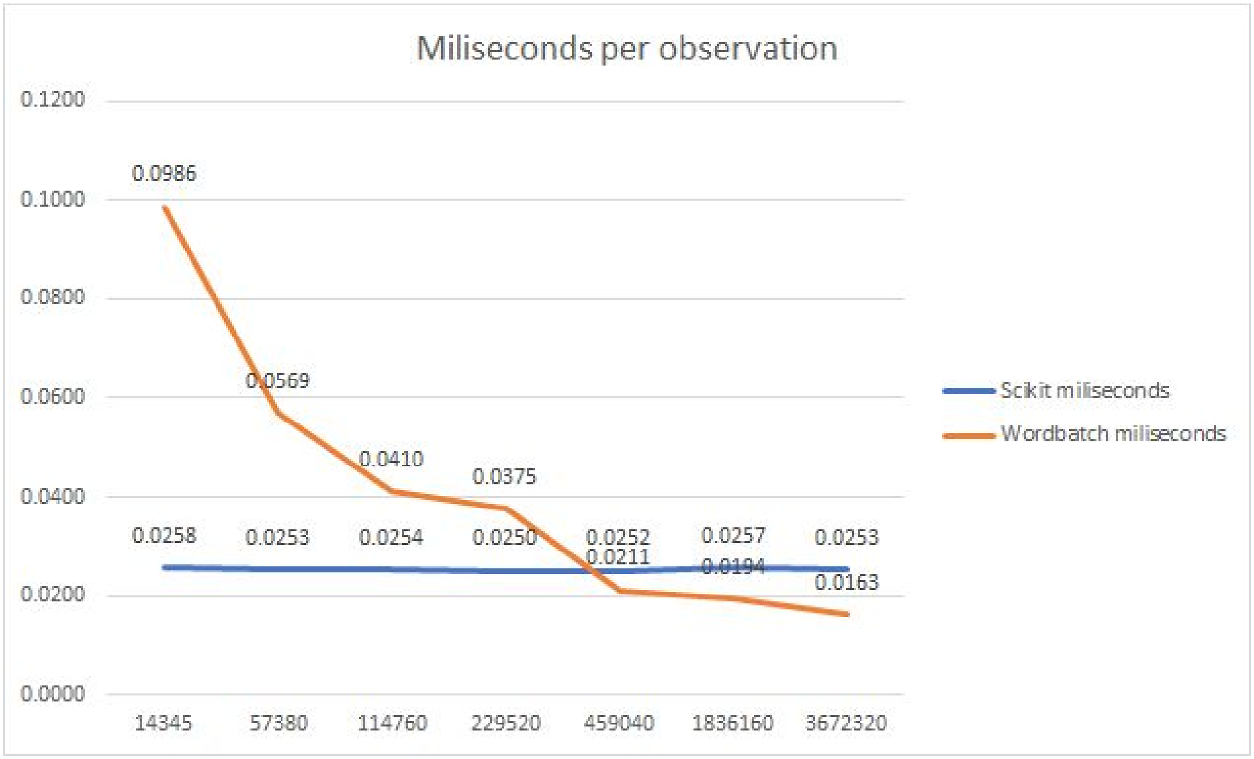
All tests will be performed both in Scikit and WordBatch for the same amount of observations and same parameters, but WordBatch will be in multiprocessing mode with 8 processors.

```
vectorizer = sklearn.feature_extraction.text.HashingVectorizer(preprocessor=normalize_text,
decode_error='ignore', n_features=2 ** 23, non_negative=False, ngram_range=(1, 2), norm='l2')

wb = wordbatch.WordBatch(normalize_text, extractor=(WordHash, {"decode_error": 'ignore',
"n_features": 2 ** 23, "non_negative": False, "ngram_range": (1, 2), "norm": 'l2'}), procs=8)
```

Nº of Observations	Processing Time Scikit-Learn	Processing Time WordBatch	Miliseconds by observation Scikit	Miliseconds by observation WordBatch
14345	0.36	5.39	0.0251	0.3757
57380	1.48	5.66	0.0258	0.0986
114760	2.9	6.53	0.0253	0.0569
229520	5.83	9.42	0.0254	0.0410
459040	11.49	17.2	0.0250	0.0375
1836160	46.2	38.75	0.0252	0.0211
3672320	94.25	71.24	0.0257	0.0194
14689280	372.12	239.22	0.0253	0.0163

*processing time is measure in seconds



The trend is clear, as you increase the observations WordBatch start to converge faster and decrease the amount of total time processing by more than 35%.

If you have a large dataset to Hash Vectorizer, WordBatch ,by far, will be faster than Scikit-Learn.

These measurement was done using a single local machine, with 8 cores. If you have a real large data you can make use of Spark just passing the spark session to the method, this can have 1000% more processing gain, depending on the amount of clusters used by Spark.

WordBag

According to the method constructor WordBag is like the WordHash but with new capabilities that does not exist in Scikit until this moment (since WordHash is built over it).
The new capabilities added to WordBag are:

- IDF (There is a TfidfVectorizer in Scikit)
- Windowed and distance-weighted polynomial interaction
- Per n-gram order weighting of hashed features
- Additional transforms for count

The parameters accepted by WordBag are:

- "norm", default value is l2
    - This is the kind of normalization you want to perform, the values be: l0, l1 or l2
- "tf", default value is 'log'
    - This is the term-frequency that can be log or binary
        - The Binary transformation execute a numpy Sign method into the data : https://docs.scipy.org/doc/numpy/reference/generated/numpy.sign.html
- "idf", default value is 0.0
    - Inverse Document-Frequency to deal with the commun words weights
    - It’s calculated for each word using the follow:
        - First calculate the normalized idf:
          `norm_idf= 1.0 / log(max(1.0, idf_parameter + doc_count))`

        - Next use the normalized idf to calculate the idf
          `idf= log(max(1.0, idf_parameter + doc_count / df)) * norm_idf`
- "hash_ngrams", default value is 0
    - Number of ngrams to hash
- "hash_ngrams_weights", default value is None
    - Interval of weights to be use for hashing
- "hash_size", default value is 10000000
    - Number of features to be generate
- "hash_polys_window", default value is 0
    - Windows to be used in polynomial interactions
- "hash_polys_mindf", default value is 5
    - Min Document Frequency to be use in polynomial
- "hash_polys_maxdf", default value is 0.5
    - Max Document Frequency to be use in polynomial
- "hash_polys_weight", default value is 0.1
    - Weight multiplier to be used in polynomial features
- "seed", default value is 0
    - Regular randomizer multiplier to keep test consistency

Here is an example of the WordBag implementation:

```
wb = wordbatch.WordBatch(normalize_text
                , extractor=(WordBag, {"hash_ngrams": 2,
                                "hash_ngrams_weights": [0.5, -1.0],
                                "hash_size": 2 ** 23,
                                "norm": 'l2',
                                "tf": 'log',
                                "idf": 50.0}
                    )
                , procs=8)
```

*normalize_text is a custom function for normalization

WordSeq

This method provide sequences of words integers, constantly used for Long Short-Term Memory models as input, for example:

We define a max length and a max number of words to be used and pass it to the WordSeq:

```
maxlen = 200
max_words = 20000
wb= wordbatch.WordBatch(normalize_text, max_words=max_words, extractor=(WordSeq,
{"seq_maxlen": maxlen}))
```

And use it to transform our input to be predict

```
self.model.predict(np.array(self.wb.transform(texts)))
```

WordVec

Word to vector is a method to create word embeddings, mapping words from a corpus into vectors of real numbers.

WordBatch allow the use of multiples Gloves files on the same call, and they will be normalized and loaded into the same stack result.

The way to use this method is the follow:

```
wb = wordbatch.WordBatch(normalize_text,
                    extractor=(Hstack,[
                      (WordVec,{"wordvec_file":
"../../../data/word2vec/glove.twitter.27B.100d.txt.gz",
                            "normalize_text": normalize_text, "encoding": "utf8"}),
                      (WordVec, {"wordvec_file": "../../../data/word2vec/glove.6B.50d.txt.gz",
                             "normalize_text": normalize_text, "encoding": "utf8"})]))
```

Next, you use the transform() function to convert the text, and send it to the be trained/predicted

```
texts= self.wb.fit_transform(texts, reset= False)
```

A list of extractors can be defined. For example, word vector sequences can be projected into per-document vectors, and concatenated with the vectors from other word vector embeddings. [1]

The Models

WordBatch not only provide extractors and features transformations, it comes with four basic Online Learning Models, for single-label regression and classification.

All four models are built to run in multi processing architectures with L1 and L2 regularizations.

FTRL (Follow the Regularized Leader)

The linear Model, Proximal-FTRL is one of the most used Kaggle competition models, created by Google (http://www.eecs.tufts.edu/~dsculley/papers/ad-click-prediction.pdf), it's implemented in Cython-optmized and is one of the fastest versions of the FTRL model available.

As in the paper, you iterate through the data and for each instance:

1. Create a feature vector
2. Compute a prediction using the current weight vector
3. Update the model

Here is a example on how to use it with WordBatch:

```
clf= FTRL(alpha=1.0, beta=1.0, L1=0.00001, L2=1.0, D=2 ** 23, iters=1,
inv_link="identity")
```

The parameters and default values, used by the model are the following:

- alpha=0.1,
- beta=1.0,
- L1=1.0,
- L2=1.0,
- D=2**25,
- init= 0.0,
- iters=10,
- e_clip= 1.0,
- threads= 0,
- inv_link= "sigmoid",
- bbias_term=1,
- seed= 0,
- verbose=1

FM_FTRL (Factorization Machines)

Hashed Factorization Machine with Follow The Regularized Leader and factor effects estimated with adaptive SGD.

```
clf = FM_FTRL(D=2 ** 25, D_fm= 4, iters=1, inv_link="identity", threads=
multiprocessing.cpu_count()//2)
```

The parameters and default values, used by the model are the following:

- alpha=0.02,
- beta=0.01, # ~ alpha/2
- L1=0.0001,
- L2=0.1,
- D=2**25,
- alpha_fm=0.03,
- L2_fm= 0.005,
- init_fm= 0.01,
- D_fm=20,
- weight_fm= 10.0,
- e_noise= 0.0001,
- e_clip= 1.0,
- iters=5,
- inv_link= "identity",
- bbias_term=1,
- threads= 0,
- use_avx=1,
- seed= 0,
- verbose=1

NN_Relu_H1

Neural Network with 1 hidden layer and Rectified Linear Unit activations, estimated with adaptive SGD.

The prediction and estimation are multithreaded across hidden layer.

The parameters and default values, used by the model are the following:

- double alpha=0.1,
- double L2=0.001,
- int D=2**25,

- int D_nn=30,
- double init_nn=0.01,
- double e_noise=0.0001,
- double e_clip=1.0,
- unsigned int iters=4,
- inv_link= "identity",
- int threads= 0,
- int seed= 0,
- int verbose=1

NN_Relu_H2

Neural Network with 2 hidden layers and Rectified Linear Unit activations, estimated with adaptive SGD.

The prediction is multithreaded across 2nd hidden layer, estimation across 1st hidden layer outputs.

The parameters and default values, used by the model are the following:

- double alpha=0.1,
- double L2=0.00001,
- int D=2**25,
- int D_nn=12,
- int D_nn2=4,
- double init_nn=0.01,
- double e_noise=0.001,
- double e_clip=1.0,
- unsigned int iters=3,
- inv_link= "identity",
- int threads= 0,
- int seed= 0,
- int verbose=1

The adaptive SGD optimizer works like Adagrad in all the models that make use of it, but pools the adaptive learning rates across hidden nodes using the same feature. This makes learning more robust and requires less memory.[1]

To conclude this technical review, we have gone through all the extractors and models present on the current WordBatch version.
Batch processing has become a must for any scalable solution nowadays.
The datasets will only be larger and larger, making the need for multi processing libraries more evident than never.

REFERENCES
[1] Wordbatch https://github.com/anttttti/Wordbatch