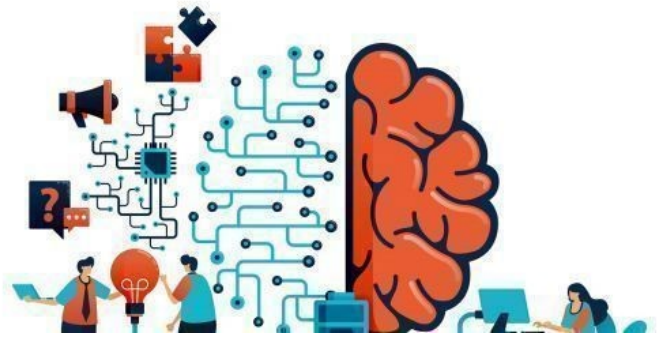
Out-of-Core ML: An Efficient Technique to Handle Large Data

INTERMEDIATE LIBRARIES MACHINE LEARNING PYTHON

This article was published as a part of the Data Science Blogathon.



Source: https://www.dyndevice.com

Introduction

We know that Machine Learning Algorithms need preprocessing of data, and this data may vary in size. If we do not have enough computing power to pre-process it, we can not proceed with analyzing it, and then the question is how to load and process such a large amount of data using the available system.

The first thing here is to know if this question is valid or whether such a situation comes or not. The answer is YES; this question is valid because nowadays, a common man also produces a large amount of data. We are living in an internet economy where we are producing data 24×7. And that's why we can say that companies have large data and need to process it, and here comes our question. How can we process it?

Following are some of the approaches that we can use to handle large amounts of data:

1. Sub Sampling
2. Cloud Computing
3. Out of Core ML

Subsampling

If we have a large dataset and want to process it, the simplest way is to use Subsampling. It uses the simple concept of reducing data. Data can be reduced by removing some of the samples or features or sometimes removing both features and samples.

Subsampling is a procedure that creates a new dataset with a given percentage size of the original sample, where sampling is performed randomly.

Pandas have a compelling method to perform subsampling data using the sample() function, which has various helpful parameters to manage the sampling process.

```
DataFrame.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None, ignore_index=False)
```

Parameters:

– **n: int (optional):** the number of items to sample

– **frac: float (optional):** the proportion (out of 1) of items to return

– **replace: bool (default False):** whether to sample with replacement (i.e., items can be sampled more than once)

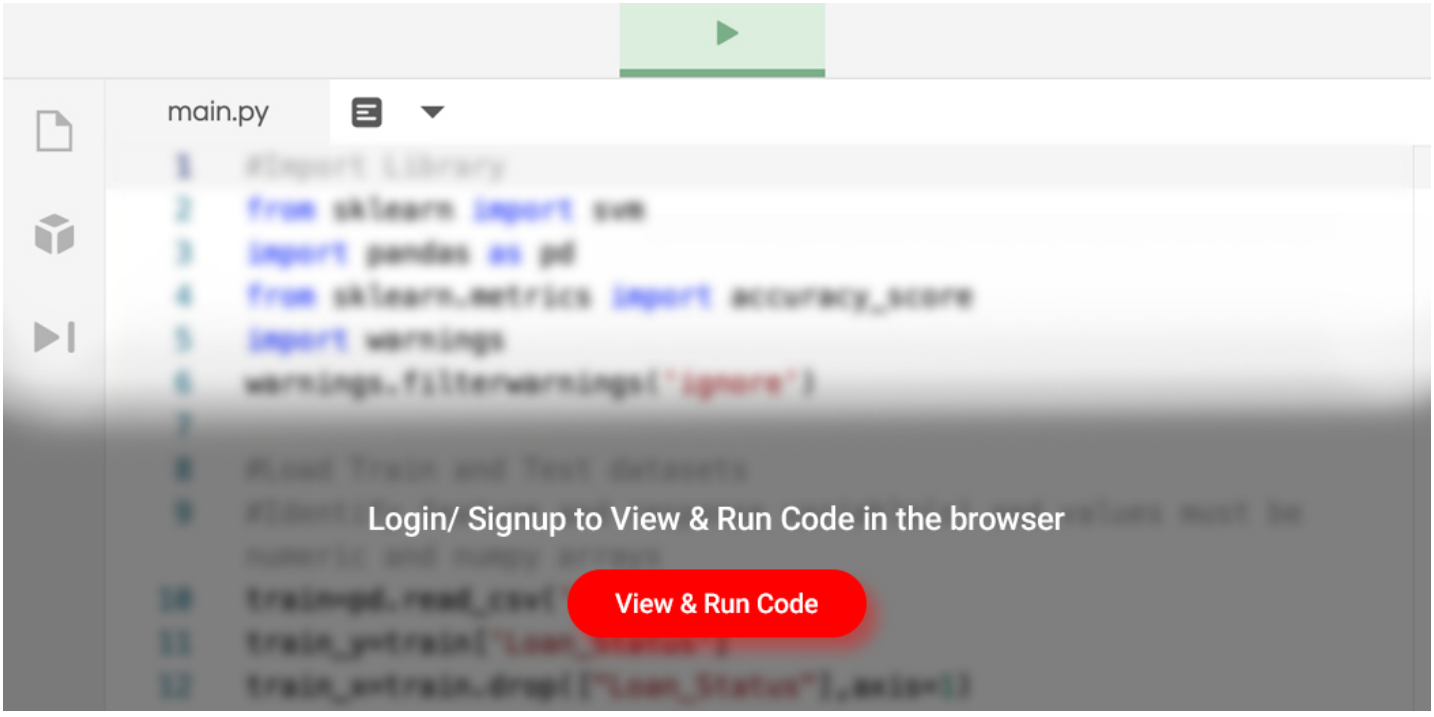
– **weight: str or ndarray (optional):** by default, samples are equally weighted. A series indicating weights can be applied. If they do not add to 1, they will be normalized to 1.

– **random_state: int:** a seed number to produce reproducible results.

– **axis: 0 or 'index', 1 or 'columns':** the axis to sample.

– **ignore_index: boolean (default False):** if we want to perform reindexing or not.

Let's see the simplest implementation of subsampling in terms of percentage using pandas on the famous titanic dataset as.



Here we can see that there are 10692 records present in our dataset. Now let's look at an example where we will pull 1% of the records using the sample's frac parameter ().

```
sample_df = df.sample(frac=0.01) print(sample_df.size) sample_df
```

```
108
```

	PassengerId	Survived	Pclass	Name	Gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
780	781	1	3	Ayoub, Miss. Banoura	female	13.0	0	0	2687	7.2292	NaN	C
826	827	0	3	Lam, Mr. Len	male	NaN	0	0	1601	56.4958	NaN	S
804	805	1	3	Hedman, Mr. Oskar Arvid	male	27.0	0	0	347089	6.9750	NaN	S
75	76	0	3	Moen, Mr. Sigurd Hansen	male	25.0	0	0	348123	7.6500	F G73	S
115	116	0	3	Pekoniemi, Mr. Edvard	male	21.0	0	0	STON/O 2. 3101294	7.9250	NaN	S
855	856	1	3	Aks, Mrs. Sam (Leah Rosen)	female	18.0	0	1	392091	9.3500	NaN	S
540	541	1	1	Crosby, Miss. Harriet R	female	36.0	0	2	WE/P 5735	71.0000	B22	S
201	202	0	3	Sage, Mr. Frederick	male	NaN	8	2	CA. 2343	69.5500	NaN	S
84	85	1	2	Ilett, Miss. Bertha	female	17.0	0	0	SO/C 14885	10.5000	NaN	S

We can see that 1% of the data frame is sampled (108 records). Here, the first column represents the index from the original data frame. We can reset it using ignore index parameter.

But the main problem with subsampling is data loss. And we know that data is essential in ML, especially when we want to build a good model.

Cloud Computing

This is another way to handle a large dataset. Anyone can use various cloud service providers like Amazon AWS, Google Cloud Platform, or Microsoft Azure.

In cloud computing, we can configure the server as per our requirement, where we can configure memory (primary memory or core memory) ranging from Gigabyte to Terabyte.

Source: https://www.educba.com

The main disadvantage of the cloud platform is its Cost.

Out-of-Core ML

Out-of-core ML takes the reference from the concept of core memory which is nothing but RAM. We can say that out-of-core memory is only external memory which may be HDD.

If we want to define what is Out of Core ML in layman's terms, it would be:

"It is a way to train your model on data that cannot fit your core memory."

Out-of-core learning refers to the machine learning algorithms working with data that cannot fit into a single machine's memory but can easily fit into some data storage, such as a local hard disk or web repository.

There are three ways to perform it in three steps:

- **a. Streaming data**

In simple terms, streaming data is nothing but loading your data in mini-batches from secondary storage like HDD or web repository into the core memory. For example, if we have 10GB of data, we can divide it into 10 batches of 1 GB each. We will load the first 1GB, process it after processing, removes it from core memory, and load the next batch.

But to do this, we need a reader. Generally, we call it a data reader. This reader facility is provided by Pandas IO library, which allows us to read data in chunks. In deep learning, Keras has built-in data readers. In Machine Learning, the **Vaex** library is specifically designed for such operation.

- ***b. Extracting features***

Once a chunk of data is in RAM or core memory, we need to extract features from that data. These features we extract from data are used as input for the ML model. This feature extraction is a little tricky part.

- ***c. Training model***

In normal conditions, we have all our data in RAM to train our model on that data. But here, we are not able to load all data in memory. Even though we extract features in the second step, those features may be larger than RAM.

Because of this, we cannot train all the data at once using a single machine learning algorithm.

Here, we can use the concept of Incremental Learning, where we train the machine learning algorithm by providing data in batches.

As we can see in the above figure, 1st chunk is given as input to Model M which generates output as M', then 2md chunk is given as input to Model M' instead of Model M. So, M' model will not train from the beginning instead it trains from M and so on.

Not all Machine Learning Algorithms support Incremental Learning. Few supported algorithms are Multinomial Naive Bayes, Bernoulli Naive Bayes, Perceptron, SGDClassifier, PassiveAggressiveClassifier, SGDRegressor, PassiveAggressiveRegerssor, MiniBatchKMeans, etc.

Coding Part

Let's see how we can implement it. We have a large dataset of 3GB and contain 30 million or 3 crores records. This is a dummy toy dataset having 4 independent features and 1 target feature, and it is a regression problem.

Please note that this is just dummy data, and we are trying to understand the out-of-core ML concept so we are not focusing on the data cleaning and other tasks.

```
import pandas as pd df = pd.read_csv('large_dataset.csv') df.shape
```

```
(30000000, 5)
```

```
df.head()
```

	f1	f2	f3	f4	target
0	0.203936	-0.291191	-0.617251	-1.461300	10.796660
1	-1.195432	0.069214	0.291861	1.972511	11.154018
2	-2.560271	-0.827846	0.405551	-0.132588	-20.342684
3	-1.131196	0.252654	0.961175	1.236873	1.407638
4	0.858774	-0.461605	1.095498	-0.871580	-17.043060

```
df.describe()
```

	f1	f2	f3	f4	target
count	3.000000e+07	3.000000e+07	3.000000e+07	3.000000e+07	3.000000e+07
mean	-3.258116e-04	-1.060974e-04	-1.360234e-05	-6.534420e-05	-8.363976e-03
std	9.999900e-01	1.000074e+00	9.998615e-01	1.000105e+00	3.131135e+01
min	-5.491886e+00	-5.333572e+00	-5.312364e+00	-5.237933e+00	-1.694348e+02
25%	-6.748877e-01	-6.744943e-01	-6.743856e-01	-6.744510e-01	-2.112416e+01
50%	-3.116059e-04	-2.164725e-04	-4.358653e-05	-2.934173e-04	-1.129371e-02
75%	6.741835e-01	6.743107e-01	6.744813e-01	6.746014e-01	2.111568e+01
max	5.500586e+00	5.449671e+00	5.500811e+00	5.347441e+00	1.778399e+02

Now, if we try to use Panda's library, it would take much time to read these records, and the process becomes very slow.

So instead of Pandas, we are going to use **Vaex** library, and we will train the ML model using this library.

First, we need to install vaex library using.

```
pip install vaex
```

```
vaex.dataframe.DataFrameLocal
```

We need to convert our dataset from csv to hdf5 file format. hdf5 file format is a hierarchal format where data is stored efficiently, and because of this, searching becomes very easy, which speeds up the operations.

```
import vaex vaex_df = vaex.from_pandas(df) type(vaex_df)
```

```
vaex.dataframe.DataFrameLocal
```

```
vaex_df.export_hdf5('large_data.hdf5') vaex_df = vaex.open('large_data.hdf5') vaex_df.head()
```

#	f1	f2	f3	f4	target
0	0.203936	-0.291191	-0.617251	-1.4613	10.7967
1	-1.19543	0.0692141	0.291861	1.97251	11.154
2	-2.56027	-0.827846	0.405551	-0.132588	-20.3427
3	-1.1312	0.252654	0.961175	1.23687	1.40764
4	0.858774	-0.461605	1.0955	-0.87158	-17.0431
5	0.63273	2.07437	-0.700978	1.36085	49.2754
6	-1.92771	0.155743	-0.394158	-1.3712	11.0902
7	-0.165462	-0.481113	-1.29813	1.75204	-41.1136
8	-1.31883	0.393728	-1.83311	-0.599728	1.87724
9	-0.671377	-0.172073	-0.75585	1.05745	-26.2623

Vaex library is fast because it reads the metadata of hdf5 data when we use vaex_open() and as per requirement, only that data will be fetched/loaded into memory; for example, if we use head() then we know that we need only first 10 rows and vaex will only fetch first 10 rows into memory. At the same time, Pandas will load all the data in memory when we use the read_csv() function.

We are not performing preprocessing. So we will directly perform train_test_split(). We need to shuffle data first before performing a split.

```
vaex_df = vaex_df.shuffle() df_train, df_test = vaex_df.ml.train_test_split(test_size=0.2) df_train.shape
```

```
(24000000, 5)
```

```
df_test.shape
```

```
(6000000, 5)
```

The next step is to train the model, and we are using incremental learning to train the model. From the sklearn package of vaex.ml module, we are importing IncrementalPredictor class.

We provide independent features (input columns, input features) and the target variable.

```
from vaex.ml.sklearn import IncrementalPredictor from sklearn.linear_model import SGDRegressor features = ['f1','f2','f3','f4'] target = 'target' model = SGDRegressor()
```

We have created the model of SGDRegressor, which we are providing as a parameter to the vaex incremental model.

For the vaex incremental model, we are passing input features, target variable, model (i.e., SGDRegressor), and the important thing which is the batch size which is nothing but our chunks for incremental learning (how many numbers of rows we are passing at a time to model for training. We are passing 5lakh rows at a time in this example.)

```
vaex_model = IncrementalPredictor(features=features, target=target, model=model, batch_size=500000) vaex_model.fit(df=df_train,progress='widget')
```

While using the vaex library transform() function performs prediction, the same task in Pandas will be made using predict() function.

Transform() will give us a data frame with one extra 'predict' column.

```
df_test = vaex_model.transform(df_test) df_test.head()
```

#	f1	f2	f3	f4	target	prediction
0	0.651098	1.75828	-1.55602	0.205312	52.0572	41.9285
1	-0.930079	0.66736	1.38524	-0.785785	17.9182	16.1285
2	-0.757777	-1.0052	1.03499	0.227977	-37.9689	-24.1394
3	0.320047	1.03793	1.27838	2.78425	-7.66551	25.0803
4	-1.57124	-0.383367	0.162823	-0.70573	-13.792	-9.57404
5	-1.25979	-0.777414	0.224138	-0.925356	21.0758	-18.9242
6	-0.0639199	-0.143284	-0.103907	-0.761464	-9.42623	-3.48351
7	0.798208	0.53451	-1.62919	-1.36567	-8.41418	12.6728
8	0.465624	-0.659685	1.04316	0.305888	-25.7486	-15.5313
9	-0.749915	-0.151375	-0.742794	-0.0452924	20.3753	-4.0625

R2 score and MAE can be calculated as:

```
from sklearn.metrics import r2_score,mean_absolute_error
print(r2_score(df_test['target'].values,df_test['prediction'].values))
print(mean_absolute_error(df_test['target'].values,df_test['prediction'].values))
```

```
0.5920774016319219
15.96033360781515
```

Conclusion

In this article, we begin with a simple problem of how to load a large amount of data for preprocessing if we do not have a powerful enough machine. Then we have seen why we may have a large amount of data.

1. Then we learned Subsampling techniques and how we can implement them.
2. Next is Cloud Computing, but the major drawback of the Cloud is its cost.
3. Finally, we learned the most feasible and important technique, i.e., Out of Core ML, and we have seen its implementation using the vaex library. Three ways to perform out-of-core ML are streaming data, extracting features, and training the model.

- Key takeaways:
  - We may get a large amount of data on whether your system can handle it.
  - There are various techniques available to handle such situations.
  - The most feasible and optimum technique is Out of Core ML and how to implement it using the vaex library.

I hope I tried to explain the concept in simple language for better understanding. I highly appreciate your feedback and comments on this article. The data set used in implementation can be found in the reference links.

References:

1. https://www.youtube.com/watch?v=sRCuvcdvuzk&t=0s

2. https://www.youtube.com/watch?v=9e4nUuq2Hmg

3. https://github.com/campusx-official/vaex-demo

4. https://scikit-learn.org/0.15/modules/scaling_strategies.html

5. https://datagy.io/pandas-sample/

Article Url - https://www.analyticsvidhya.com/blog/2022/09/out-of-core-ml-an-efficient-technique-to-handle-large-data/

Shekhar Kausalye