MAKING SENSE OF BIG DATA

How to Work with Million-row Datasets Like a Pro

It is time to take off your training wheels

Bex T. Sep 8, 2021 · 8 min read ⋆

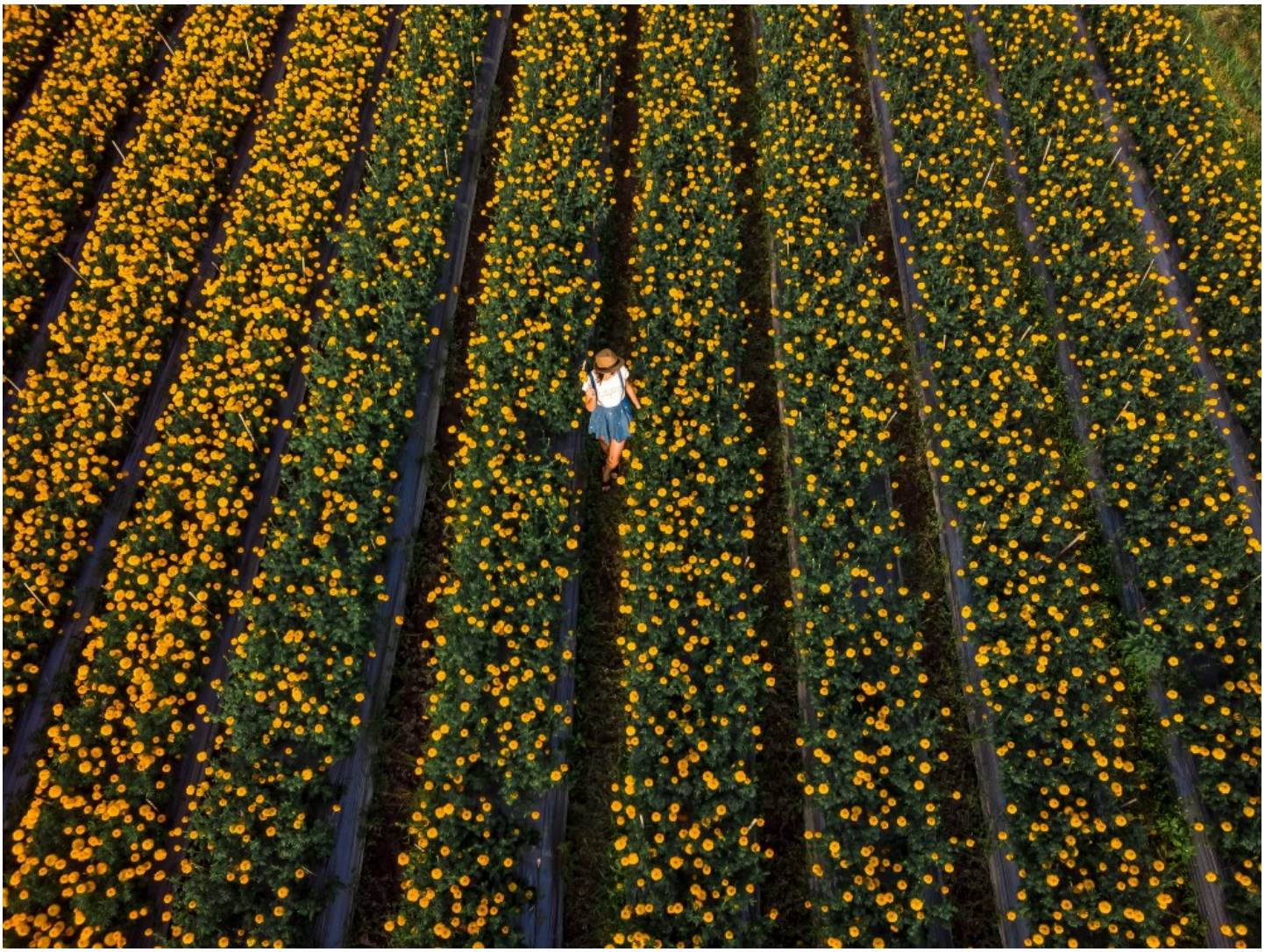


Photo by Artem Beliaikin on Pexels. All images are by the author unless specified otherwise.

Introduction

One of the difficult stages of my learning journey was about overcoming my fear of massive datasets. It wasn't easy because working with million-row

datasets was nothing like the tiny, toy datasets the online courses continuously gave me.

Today, I am here to share the concepts and tricks I have learned to handle the challenges of gigabyte-sized datasets with millions or even billions of rows. By the end, they will feel to you almost as natural as working with the Iris or Titanic.

The notebook of this article can be found on Kaggle here or from this repo.

Read in the massive dataset

The first of your worries start when loading the data — the time it takes to read the dataset into your working environment can be as long as you train a model. At this stage, don't use pandas — there are much faster alternatives available. One of my favorites is the `datatable` package which can read data up to 10 times faster.

As an example, we will load ~1M row Kaggle TPS September 2021 dataset with both `datatable` and `pandas` and compare the speeds:

```
import datatable as dt  # pip install datatable
import pandas as pd

%%time
tps_dt = dt.fread("data/tps_september_train.csv").to_pandas()
tps_dt.head()

----------------------------------

Wall time: 3.01 s
```

9001.py hosted with ❤ by GitHub view raw

	id	f1	f2	f3	f4	f5	f6	f7	f8	f9	...	f110	f111	f112	f113	f114	f115	f116	f117	f118	claim
0	0	0.10859	0.004314	-37.566	0.017364	0.28915	-10.25100	135.12	168900.0	3.992400e+14	...	-12.2280	1.7482	1.90960	-7.11570	4378.80	1.2096	8.613400e+14	140.1	1.01770	True
1	1	0.10090	0.299610	11822.000	0.276500	0.45970	-0.83733	1721.90	119810.0	3.874100e+15	...	-56.7580	4.1684	0.34808	4.14200	913.23	1.2464	7.575100e+15	1861.0	0.28359	False
2	2	0.17803	-0.006980	907.270	0.272140	0.45948	0.17327	2298.00	360650.0	1.224500e+13	...	-5.7688	1.2042	0.26290	8.13120	45119.00	1.1764	3.218100e+14	3838.2	0.40690	True
3	3	0.15236	0.007259	780.100	0.025179	0.51947	7.49140	112.51	259490.0	7.781400e+13	...	-34.8580	2.0694	0.79631	-16.33600	4952.40	1.1784	4.533000e+12	4889.1	0.51486	True
4	4	0.11623	0.502900	-109.150	0.297910	0.34490	-0.40932	2538.90	65332.0	1.907200e+15	...	-13.6410	1.5298	1.14640	-0.43124	3856.50	1.4830	-8.991300e+12	NaN	0.23049	True

5 rows × 120 columns

Source: Kaggle TPS September Competition

```
%%time

tps_df = pd.read_csv("data/tps_september_train.csv")
tps_df.head()

-----------------------------------------------

Wall time: 23.8 s
```

9002.py hosted with ❤ by GitHub view raw

	id	f1	f2	f3	f4	f5	f6	f7	f8	f9	...	f110	f111	f112	f113	f114	f115	f116	f117	f118	claim
0	0	0.10859	0.004314	-37.566	0.017364	0.28915	-10.25100	135.12	168900.0	3.992400e+14	...	-12.2280	1.7482	1.90960	-7.11570	4378.80	1.2096	8.613400e+14	140.1	1.01770	1
1	1	0.10090	0.299610	11822.000	0.276500	0.45970	-0.83733	1721.90	119810.0	3.874100e+15	...	-56.7580	4.1684	0.34808	4.14200	913.23	1.2464	7.575100e+15	1861.0	0.28359	0
2	2	0.17803	-0.006980	907.270	0.272140	0.45948	0.17327	2298.00	360650.0	1.224500e+13	...	-5.7688	1.2042	0.26290	8.13120	45119.00	1.1764	3.218100e+14	3838.2	0.40690	1
3	3	0.15236	0.007259	780.100	0.025179	0.51947	7.49140	112.51	259490.0	7.781400e+13	...	-34.8580	2.0694	0.79631	-16.33600	4952.40	1.1784	4.533000e+12	4889.1	0.51486	1
4	4	0.11623	0.502900	-109.150	0.297910	0.34490	-0.40932	2538.90	65332.0	1.907200e+15	...	-13.6410	1.5298	1.14640	-0.43124	3856.50	1.4830	-8.991300e+12	NaN	0.23049	1

5 rows × 120 columns

7 times speedup! The `datatable` API for manipulating data may not be as intuitive as `pandas` - so, call the `to_pandas` method after reading the data to

convert it to a DataFrame.

Apart from `datatable`, there are `Dask`, `Vaex`, or `cuDF`, etc. that read data multiple times faster than pandas. If you want to see some of those in action, refer to this notebook on reading large datasets by Kaggle Grandmaster Rohan Rao.

Reduce the memory size

Next, we have memory issues. Even a 200k row dataset may exhaust your 16GB RAM while doing complex computations.

I have experienced this first-hand *twice* in the last month's TPS competition on Kaggle. The first one was when projecting the training data to 2D using UMAP — I ran out of RAM. The second was while computing the SHAP values with XGBoost for the test set — I ran out of GPU VRAM. What is shocking is that the training and test sets only had 250k and 150k rows with a hundred features, and I was using Kaggle kernels.

The dataset we are using today has ~960k rows with 120 features, so memory issues are much more likely:

```
memory_usage = tps_df.memory_usage(deep=True) / 1024 ** 2

>>> memory_usage.head(7)
Index    0.000122
id       7.308342
f1       7.308342
f2       7.308342
f3       7.308342
f4       7.308342
f5       7.308342
dtype: float64

---------------------------------------

>>> memory_usage.sum()
877.0011596679688
```

9003.py hosted with ❤ by GitHub view raw

Using the `memory_usage` method on a DataFrame with `deep=True`, we can get the exact estimate of how much RAM each feature is consuming - 7 MBs. Overall, it is close to 1GB.

Now, there are certain tricks you can use to decrease memory usage up to 90%. These tricks have a lot to do with changing the data type of each feature to the smallest subtype possible.

Python represents various data with unique types such as `int`, `float`, `str`, etc. In contrast, pandas has several NumPy alternatives for each of Python's:

Pandas dtype	Python type	NumPy type	Usage
object	str	string_, unicode_	Text
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64	Integer numbers
float64	float	float_, float16, float32, float64	Floating point numbers
bool	bool	bool_	True/False values

datetime64	NA	datetime64[ns]	Date and time values
timedelta[ns]	NA	NA	Differences between two datetimes
category	NA	NA	Finite list of text values

Source: http://pbpython.com/pandas_dtypes.html

Numbers next to the datatype refer to how many bits of memory a single data unit consumes when represented in that format. To reduce the memory as much as possible, choose the smallest NumPy data format. Here is a good table to understand this:

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Source: https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html

In the above table, `uint` refers to unsigned, only positive integers. I have found this handy function that reduces the memory of pandas DataFrames based on the above table (shout out to this Kaggle kernel):

```
def reduce_memory_usage(df, verbose=True):
    numerics = ["int8", "int16", "int32", "int64", "float16", "float32", "float64"]
    start_mem = df.memory_usage().sum() / 1024 ** 2
    for col in df.columns:
        col_type = df[col].dtypes
        if col_type in numerics:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == "int":
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if (
                    c_min > np.finfo(np.float16).min
                    and c_max < np.finfo(np.float16).max
                ):
                    df[col] = df[col].astype(np.float16)
                elif (
                    c_min > np.finfo(np.float32).min
                    and c_max < np.finfo(np.float32).max
                ):
                    df[col] = df[col].astype(np.float32)
                else:
```

```
                    df[col] = df[col].astype(np.float64)
    end_mem = df.memory_usage().sum() / 1024 ** 2
    if verbose:
        print(
            "Mem. usage decreased to {:.2f} Mb ({:.1f}% reduction)".format(
                end_mem, 100 * (start_mem - end_mem) / start_mem
            )
        )
    return df
```

9004.py hosted with ❤ by GitHub view raw

Based on the minimum and maximum value of a *numeric* column and the above table, the function converts it to the smallest subtype possible. Let's use it on our data:

```
>>> reduced_df = reduce_memory_usage(tps_df, verbose=True)
Mem. usage decreased to 262.19 Mb (70.1% reduction)
```

9005.py hosted with ❤ by GitHub view raw

70% memory reduction is pretty impressive. However, please note that memory reduction won't speed up computation in most cases. If the memory size is not an issue, you can skip this step.

Regarding non-numeric data types, never use the `object` datatype in Pandas as it consumes the most memory. Either use `str` or `category` if there are few unique values in the feature. In fact, using `pd.Categorical` data type can speed things up to 10 times while using LightGBM's default categorical handler.

For other data types like `datetime` or `timedelta`, use the native formats offered in `pandas` since they enable special manipulation functions.

Choose a data manipulation library

Up until this point, I mainly mentioned `pandas`. It might be slow, but the vast range of data manipulation functions gives it a mounting advantage over its competitors.

But what can its competitors do? Let's start with `datatable` (again).

`datatable` allows multi-threaded preprocessing of datasets sized up to 100 GBs. At such scales, `pandas` starts throwing memory errors while `datatable` humbly executes. You can read this excellent article by Parul Pandey for an intro to the package.

Another alternative is `cuDF`, developed by RAPIDS. This package has many dependencies and can be used in extreme cases (think hundreds of billions). It enables running preprocessing functions distributed over one or more GPUs, as is the requirement by most of today's data applications. Unlike `datatable`, its API is very similar to `pandas`. Read this article from the NVIDIA blog for more information.

You can also check out Dask or Vaex that offer similar functionalities.

If you are dead set on `pandas`, then read on to the next section.

Sample the data

Regardless of any speed tricks or packages on GPU steroids, too much data, well, is too much. When you have millions of rows, there is a good chance you can sample them so that all feature distributions are preserved.

This is done mainly to speed up computation. Take a small sample instead of running experiments, feature engineering, and training baseline models on all the data. Typically, 10–20% is enough. Here is how it is done in pandas:

```
sample_df = tps_df.sample(int(len(tps_df) * 0.2))
sample_df.shape
(191583, 120)
```

9006.py hosted with ❤ by GitHub view raw

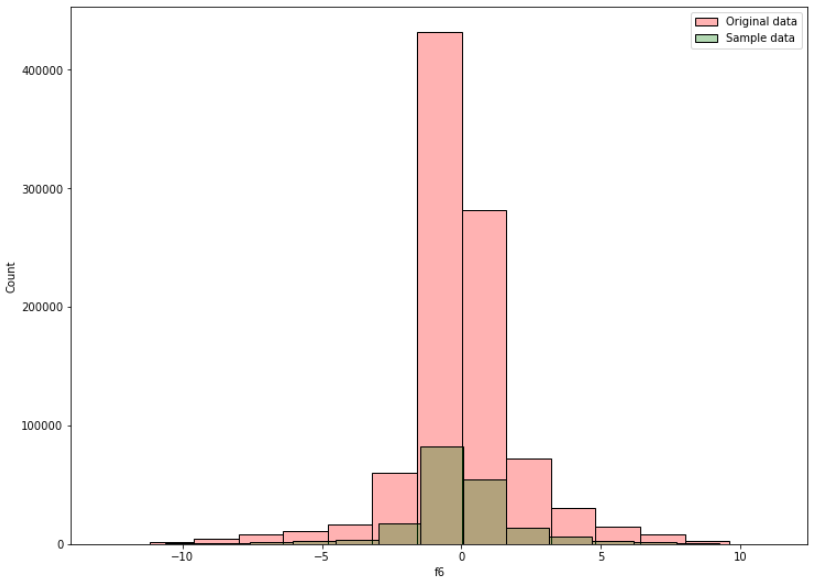
As proof, we can plot a histogram of a single feature from both the sample and the original data:

```
fig, ax = plt.subplots(figsize=(12, 9))

sns.histplot(
    data=tps_df, x="f6", label="Original data", color="red", alpha=0.3, bins=15
)
sns.histplot(
    data=sample_df, x="f6", label="Sample data", color="green", alpha=0.3, bins=15
)

plt.legend()
plt.show();
```

9007.py hosted with ❤ by GitHub view raw



As you can see, the distributions are roughly the same — you can even compare the variances to check.

Now, you can use this sample for rapid prototyping, experimenting, building a model validation strategy, and so on.

Use vectorization instead of loops

Whenever you find yourself itching to use some looping function like `apply`, `applymap`, or `itertuples` - stop. Use vectorization instead.

First, start thinking about DataFrame columns as giant n-dimensional vectors. As you know, vector operations affect each element in the vector simultaneously removing the need for loops in math. Vectorization is the process of executing operations on arrays rather than individual scalars.

Pandas has a large collection of vectorized functions. In fact, virtually any function and operator with the ability to affect each element in the array is vectorized in pandas. These functions are orders of magnitude faster than anything that loops.

You can also define custom vectorized preprocessing functions that accept whole DataFrame columns as vectors rather than scalars. The hairy details of this are beyond the scope of this article. Why don't you check out this awesome guide?

Choose a machine learning library for baseline models or prototypes

Machine learning is an iterative process. When dealing with large datasets, you have to make sure each iteration is as fast as possible. You want to build baselines, develop a validation strategy, check if different feature engineering ideas improve the baseline, and so on.

At this stage, don't use models in Sklearn because they are CPU-only. Choose from XGBoost, LightGBM or CatBoost. And here is the surprising fact — XGBoost is much slower than the other two, even on GPUs.

It is up to 10 times slower than LightGBM. CatBoost beats both libraries, and the speed difference grows rapidly as the dataset size gets bigger. It also regularly outperforms them in terms of accuracy.

These speed differences become much more pronounced when you are running multiple experiments, cross-validating, or hyperparameter tuning.

Miscellaneous tips

Use Cython (C Python) — usually, it is up to 100 times faster than pure Python. Check out this section of the Pandas documentation.

If you really have to loop, decorate your custom functions with `@numba.jit` after installing Numba. JIT (just-in-time) compilation converts pure Python to native machine instructions, enabling you to achieve C, C++, and Fortran-like speeds. Again, check this section from the docs.

Search for alternatives other than CSV files for storage. File formats like feather, parquet, and jay are lighting fast — it only takes seconds to load billion-row datasets if stored in these.

Read the Pandas documentation on enhancing performance and scaling to large datasets.

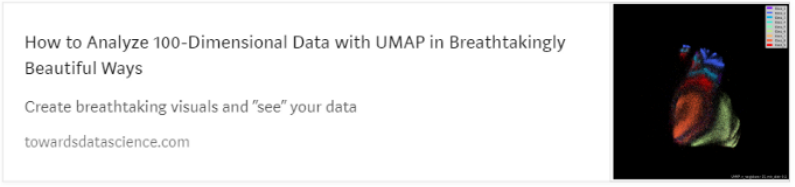
Wrapping up...

Here is a summary of the article:

1. Load the data only using libraries like `datatable`, `cuDF` or `dask`. They are always faster than Pandas.
2. Reduce the memory consumption by up to 90% by casting each column to the smallest subtype possible.
3. Choose a data manipulation library you are comfortable with or based on what you need.
4. Take a 10–20% sample of the data for rapid analysis and experimentation.
5. Think in vectors and use vectorized functions.
6. Choose a fast ML library like CatBoost for building baselines and doing feature engineering.

Thank you for reading!

You might also be interested...



Kaggler's Guide to LightGBM Hyperparameter Tuning with Optuna in 2021

Edit description

towardsdatascience.com

You Are Missing Out on LightGBM. It Crushes XGBoost in Every Aspect

Edit description

towardsdatascience.com

Tired of Cliché Datasets? Here are 18 Awesome Alternatives From All Domains

Edit description

towardsdatascience.com

Love 3Blue1Brown Animations? Learn How to Create Your Own in Python in 10 Minutes

Edit description

towardsdatascience.com