Member-only story

Advanced Pandas Techniques for Data Processing and Performance

Hands-on approach from chunked processing to parallel execution

Pratheesh Shivaprasad · Follow

Published in Towards Data Science · 7 min read · Jan 9, 2025

Photo by Weichao Deng on Unsplash

Pandas is a must-have Python library if you're working with data. Whether you're a programmer, data scientist, analyst, or researcher, you'll find it makes handling structured data much easier. It gives you flexible, intuitive tools to work with even the most complex datasets.

As you dive deeper into Pandas, mastering it can significantly boost your productivity and streamline your workflow. We will be exploring 7 essential tips that will help you leverage the library's full potential and tackle data challenges more effectively.

To illustrate the following tips, I'll be using a dataset from Kaggle's Airbnb listings. You can fetch the dataset here. *(License: CC0: Public Domain)* This dataset comprises three CSV files: `calendar.csv`, `listings.csv`, and `reviews.csv`. These files contain information about property availability, detailed listing attributes, and user reviews, respectively. By working with real-world data, I'll demonstrate how Pandas can efficiently handle and analyze complex, multi-file datasets typically encountered in data science projects.

1. Reading data in chunks

There are often scenarios we encounter where the size of the data is more than the available memory (RAM) we have. In such cases, it's a good idea to read data in chunks from the file so that the system doesn't run out of memory. This approach allows you to process data incrementally, reducing memory usage and potentially speeding up your workflow.

```
# Initialize an empty list to store the chunks
private_rooms = []

# Read the CSV file in chunks
for chunk in pd.read_csv('data/listings.csv', chunksize=1000):
    # Process each chunk (for example, filter listings where room_type is "Private room")
    processed_chunk = chunk[chunk["room_type"] == "Private room"]

    # Append the processed chunk to the list
    private_rooms.append(processed_chunk)

# Combine all processed chunks into a dataframe
private_rooms = pd.concat(private_rooms)
```

read_chunks.py hosted with ❤ by GitHub view raw

2. Render a progress bar while using the apply() function

I have always been a user of the `tqdm` library. For those who don't know about `tqdm`, it is a popular choice for displaying progress bars during iterative operations. It provides a seamless way to track the progress of loops and other iterable processes, which is particularly helpful when working with large datasets. However, I encountered a challenge in applying `tqdm` to Pandas' `apply()` functions, which are commonly used to perform element-wise operations on DataFrame columns or rows.

To my delight, I discovered that `tqdm` does, in fact, support Pandas' `apply()` methods. All it takes is a small addition to your code. Instead of using the standard `apply()` function, you can simply replace it with `progress_apply()`. This will automatically render a progress bar, giving you valuable insights into the execution of your `apply()` operations. Before that, just make sure to import `tqdm` and add a line of code below it to integrate it with pandas.

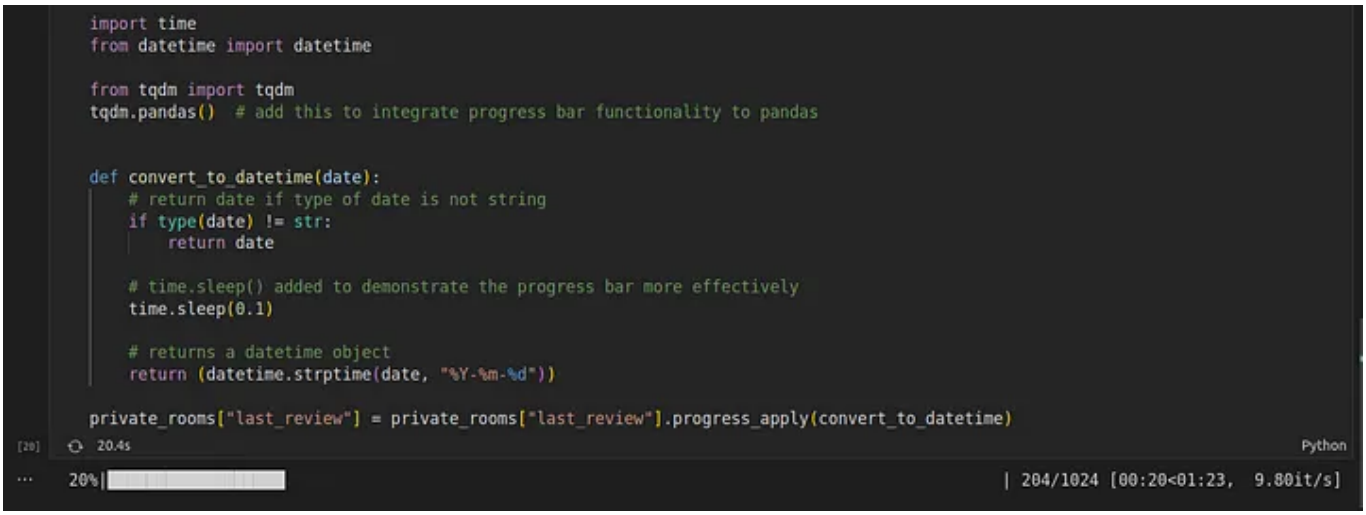
For demonstration, let's go back to the `listings.csv` dataset and convert the column `last_review` from type `string` to `datetime`. I'll use the progress_apply function instead which will create a progress bar.

If you don't have `tqdm` installed, you can use the command below to install it:

```
pip3 install tqdm
```

```python
import time
from datetime import datetime

from tqdm import tqdm
tqdm.pandas()  # add this to integrate progress bar functionality to pandas


def convert_to_datetime(date):
    # return date if type of date is not string
    if type(date) != str:
        return date

    # time.sleep() added to demonstrate the progress bar more effectively
    time.sleep(0.1)

    # returns a datetime object
    return datetime.strptime(date, "%Y-%m-%d")

private_rooms["last_review"] = private_rooms["last_review"].progress_apply(convert_to_datetime)
```

progress_apply_demo.py hosted with ❤ by GitHub view raw



Progress bar displayed at the end of the cell (Screenshot by author)

3. Effortlessly Populate Multiple Columns in Pandas Using apply() with `result_type="expand"`

While using the apply function, I often used to encounter scenarios where I need to return multiple values from a function simultaneously and store

these values in separate columns. That's when I discovered the `result_type="expand"` parameter which would help me do it.

It was such a time-saver! I no longer had to create separate functions and iterate over the same set of values just to store them in a separate column. I could just create a single function and return multiple values.

In the example below, I have demonstrated the same. I'm returning two values — the day and month name based on the review date. These values are then populated simultaneously in the `Day` and `Month` columns respectively.

```
def get_date_and_month_name(last_review_date):
    # returns the day and month
    return last_review_date.day_name(), last_review_date.month_name()


private_rooms[["Day", "Month"]] = private_rooms.apply(
    lambda x : get_date_and_month_name(x["last_review"]),
    axis=1,
    result_type="expand"
)
```

expand_result_demo.py hosted with ❤ by GitHub view raw

```
private_rooms[["name", "last_review", "Day", "Month"]]
✓ 0.0s
```

	name	last_review	Day	Month
1	Glorious sun room w/ memory foambed	2024-06-09	Sunday	June
9	"The 5-Star House" #9 The red room	2023-08-12	Saturday	August
10	Serene Room in Sunny Cottage near Discovery Park	2024-04-28	Sunday	April
11	Sunny Parisian room cozy memory foam bed	2024-06-21	Friday	June
24	art loft downtown~ID~Stadiums	2024-06-10	Monday	June
...	...	...	...	...
6382	Getaway Room - 5 min to Seattle Center	2024-06-15	Saturday	June
6387	The Sunrise Room - 10min to Downtown by Light ...	2024-06-14	Friday	June
6390	The Cherry Blossom Room - 10min to Downtown	2024-06-12	Wednesday	June
6391	The Nautical Room- 10min to Downtown by Light ...	2024-06-13	Thursday	June
6411	The Sunflower Room- 10min to Downtown	2024-06-23	Sunday	June

868 rows × 4 columns

Day and Month columns populated simultaneously (Screenshot by author)

4. Processing a DataFrame in Parallel

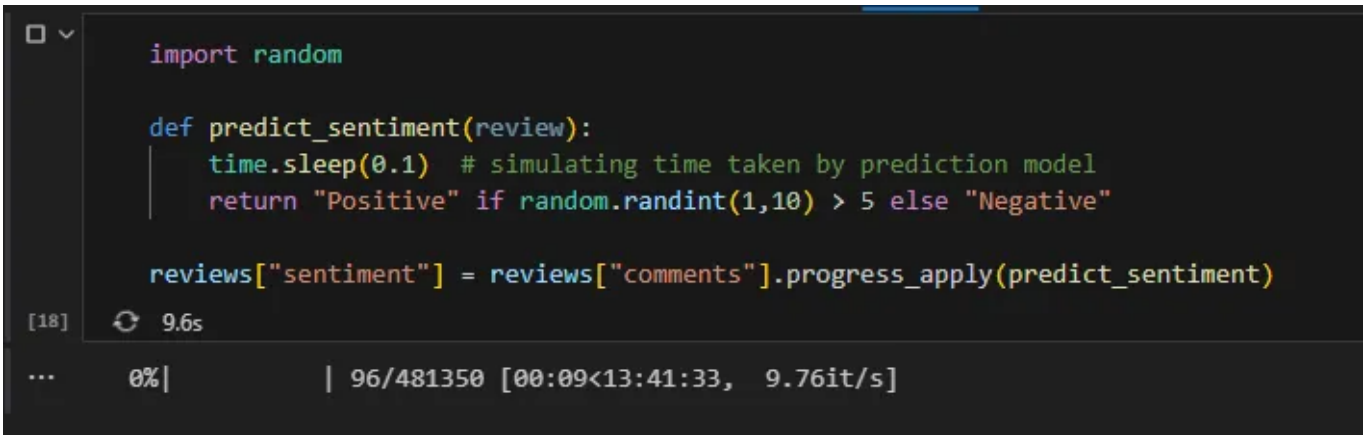
I have often faced scenarios where certain processes take too long to complete. When a DataFrame has millions of rows and you need to iterate through each row to extract certain information, it can get slow real quick.

This is where I use the multiprocessing module. I split the DataFrames (using np.array_split()) into multiple chunks (depending on the number of cores available in your system) and process each chunk of the DataFrame in parallel. This can especially be useful in cases where the system has multiple cores which most modern systems are capable of.

Once the chunks are processed and the desired output for each chunk is obtained, it can be concatenated back to a single DataFrame.

Let's use the reviews dataset for this one. It has more than 480k reviews by different users across listings.

For demonstration purposes, we will be creating a function that will simulate the time required for sentiment prediction of a review assuming each prediction will take 0.1 second.



Processing Serially (Screenshot by Author)

As you can see, it will take more than 13 hours if we make predictions one by one!

Let's speed this up. First, let's split the reviews DataFrame into multiple batches so that each batch is processed by a separate core. We can then create a multiprocessing Pool to distribute the computation over multiple cores.

```python
import time
import random
from multiprocessing import Pool

import numpy as np
import pandas as pd
from tqdm import tqdm

tqdm.pandas()


def predict_sentiment(review):
    time.sleep(0.1)  # simulating time taken by prediction model
    return "Positive" if random.randint(1,10) > 5 else "Negative"


def batch_predict_sentiment(review_df):
    review_df["sentiment"] = review_df["comments"].progress_apply(predict_sentiment)
    return review_df


def fetch_sentiment_for_review():
    n_cores = 64
    reviews = pd.read_csv("data/reviews.csv")
    review_batches = np.array_split(reviews, n_cores)  # split into same number of batches as n_c

    # Processing Parallely
    with Pool(n_cores) as pool:
        sentiment_prediction_batches = pool.map(batch_predict_sentiment, review_batches)

    # Once all the batches are processed, concatenate list of DataFrames into a single DataFrame
    sentiment_prediction = pd.concat(sentiment_prediction_batches)
    return sentiment_prediction


reviews_with_sentiment = fetch_sentiment_for_review()
```

multiprocessing_dataframes.py hosted with ❤ by GitHub view raw

```
1%|          | 58/7521 [00:05<12:46,  9.74it/s]
```

Processing in Parallel (Screenshot by Author)

Using multiprocessing, we have managed to bring down the prediction time required from more than 13 hours to less than 13 minutes!

Each batch consists of 7521 reviews and there are a total of 64 batches. In this scenario, I was able to set `n_cores` more than the actual number of cores my system has. This is because during the execution of `time.sleep(0.1)` the CPU remains idle and each process interleaves for other process to execute. If your process is CPU intensive, it is recommended to keep `n_cores` less than the actual number of cores your system has.

5. Perform Complex Merging with merge() and indicator=True

Merging is quite a common operation performed by individuals who deal with data. However, sometimes it can get quite complicated to understand if any particular data points were lost during the merging process. It might be due to a plethora of reasons — the worst one being, malformed or faulty data.

This is where the `indicator=True` parameter comes in handy. When you enable this parameter, it creates a new column named `_merge` which can denote three different scenarios based on the type of merge operation performed.

1. *left_only* — indicates that the row's key only exists in the left DataFrame and it couldn't find a match in the right DataFrame

2. *right_only* — indicates that the row's key only exists in the right

was successfully merged.

These values can also come in handy as a filter to apply during data manipulation tasks.

In the example below, I'll be performing an outer merge between reviews and listing DataFrames.

I'm performing an outer merge with the parameter `indicator=True` to identify which listings have no reviews/missing reviews. Ideally, listings with the parameter `_merge` set to *"left_only"* will be missing reviews.

```
merged_df = listings.merge(
    reviews,
    left_on="id",
    right_on="listing_id",
    how="outer",
    indicator=True
)
merged_df.shape
```

```
merged_df["_merge"].value_counts()
✓ 0.0s

_merge
both          481350
left_only        841
right_only         0
Name: count, dtype: int64
```

There are 841 listings with no reviews (Screenshot by Author)

Below are some examples of listings with no reviews

```
listings_with_no_reviews = merged_df[merged_df["_merge"] == "left_only"]
listings_with_no_reviews[["id_listings", "name", "host_name", "reviewer_id", "reviewer_name", "comments", "_merge"]].head()
```

✓ 0.0s

	id_listings	name	host_name	reviewer_id	reviewer_name	comments	_merge
4364	340738	Victorian home on Capitol Hill	Marlow	NaN	NaN	NaN	left_only
56570	4630355	Luxury Penthouse in Seattle	Bhuwan	NaN	NaN	NaN	left_only
110072	10834487	Corner room	Savannah	NaN	NaN	NaN	left_only
112097	11254431	Master suite in Ballard home	Sarah	NaN	NaN	NaN	left_only
126849	13081598	Urban Classic Lake Union & City center	Kaela	NaN	NaN	NaN	left_only

Examples of listings with no reviews (Screenshot by Author)

6. Segmenting Data into Price Brackets Using pd.cut()

pd.cut() is a powerful function that can be used when you need to segment data into multiple bins. It can also act as a way to convert continuous values into categorical values.

One such scenario is demonstrated in the example below. We will be Segmenting the price of each listing into multiple price brackets (bins).

We can set a predetermined number of price brackets — “$0 — $100”, “$101 — $250”, “$251 — $500”, “$500 — $1000”, and “$1000+”.

```
# Create bins and label for each bin
bins = [0, 100, 250, 500, 1000, float('inf')]
labels = ["$0 - $100", "$101 - $250", "$251 - $500", "$500 - $1000", "$1000+"]

listings["price_bucket"] = pd.cut(listings["price"], bins=bins, labels=labels)
```

Please note here that the number of labels (5) is less than number of bins (6) by one. This is because the initial two values in the bin belong to the first label.

```
listings["price_bucket"].value_counts().reset_index().sort_values(by="price_bucket")
```

✓ 0.0s

	price_bucket	count
2	$0 - $100	1117
0	$101 - $250	3400
1	$251 - $500	1241
3	$500 - $1000	224
4	$1000+	29

Breakup of Listing across Price Brackets (Screenshot by Author)

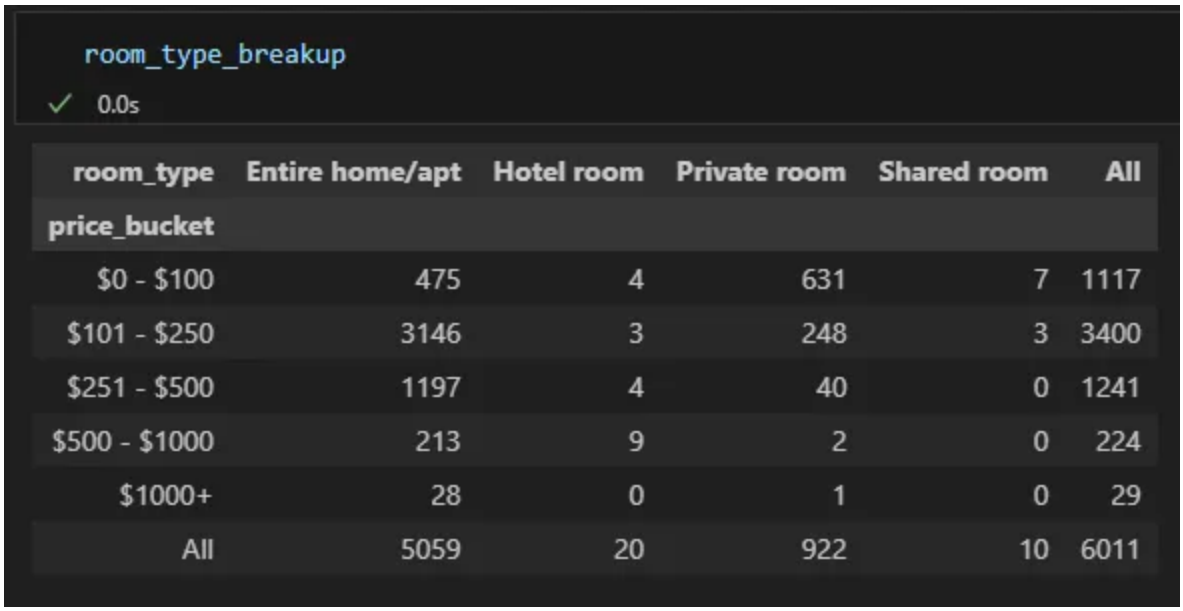
We can see that the majority of the listings (3400) lie in the range between $101–$250 with the least amount of listings (29) in the $1000+ range.

7. Cross-Tabulation Analysis of Prices and Room Types

Using the above data, we can go one step further and perform a cross-tabulation between the price brackets and room types available for those price brackets.

Here's where pd.crosstab() comes into play. It can be used to perform a simple cross-tabulation between `price_bucket` and `room_type`.

```
room_type_breakup = pd.crosstab(
    listings["price_bucket"],
    listings["room_type"],
    margins=True  # This will add the row and column totals
)
```



room_type	Entire home/apt	Hotel room	Private room	Shared room	All
price_bucket					
$0 - $100	475	4	631	7	1117
$101 - $250	3146	3	248	3	3400
$251 - $500	1197	4	40	0	1241
$500 - $1000	213	9	2	0	224
$1000+	28	0	1	0	29
All	5059	20	922	10	6011

Room Type Breakup for each Price Bracket (Screenshot by Author)

The above screenshot shows the distribution of room types across different price brackets. The last row and column will be the sum of row and column totals. Set `margins=False` if it's not desired.

By now, you've learned several powerful Pandas techniques that will significantly improve your data processing workflow. From managing large datasets with chunked reading to speeding up operations through parallel processing, these advanced methods will help you handle complex data tasks more efficiently.

Please do let me know if you recently found any other interesting techniques that improved your workflow or if you found a more efficient way to handle the above techniques!

Pandas | Data Processing | Data Analysis | Optimization | Parallel Computing

Published in Towards Data Science

800K Followers · Last published 14 hours ago

Follow

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Written by Pratheesh Shivaprasad

178 Followers · 15 Following

Follow

SDE @Intelligence Node. LinkedIn: linkedin.com/in/pratheesh-s

Responses (1)

What are your thoughts?

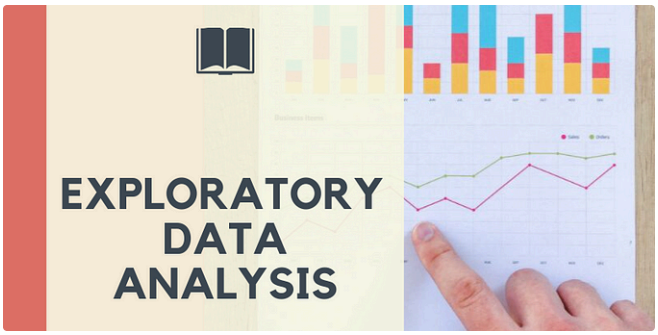
Respond

Torben Windler
Jan 16

The heavy use of apply in this article is concerning - it's essentially a slow for loop and not suitable for efficient data processing in Pandas. Readers shouldn't take this approach for granted. Always prefer vectorized operations in Pandas for... more

6 1 reply Reply

More from Pratheesh Shivaprasad and Towards Data Science

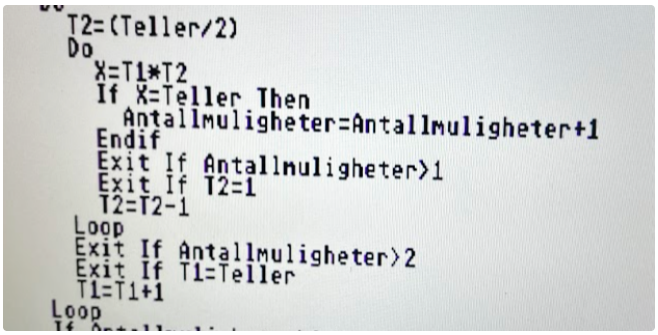


In Towards Data Science by Pratheesh Shivaprasad

Exploratory Data Analysis

Includes a simple case study for better understanding.

Aug 15, 2019 193 2



In Towards Data Science by Ragnvald Larsen

Are Public Agencies Letting Open-Source Software Down?

Open-source software is everywhere, yet public agencies and institutions often fall...

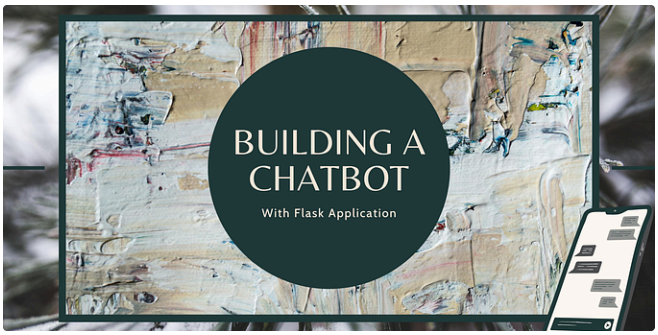
21h ago 6

In Towards Data Science by Muhammad Ardi

Show and Tell

Implementing one of the earliest neural image caption generator models with PyTorch.

23h ago 11



In Towards Data Science by Pratheesh Shivaprasad

How To Build a Basic Chatbot From Scratch

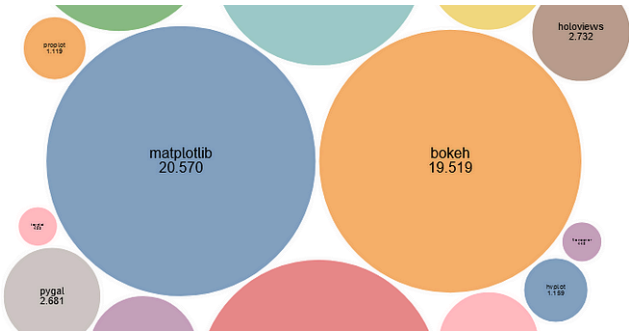
From text preprocessing, building models to a full-fledged Flask web application with AJAX

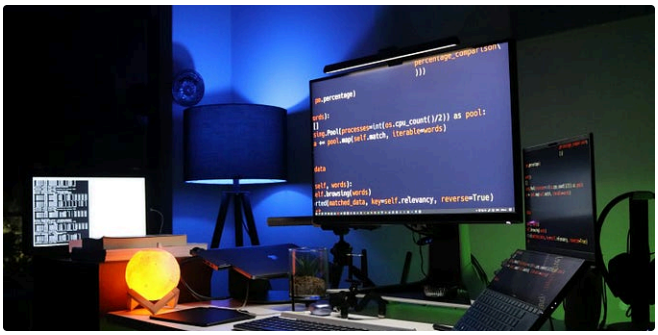
Aug 20, 2020 156 3

See all from Pratheesh Shivaprasad

See all from Towards Data Science

Recommended from Medium





In Python in Plain English by Zlatan B

Python Packages for Data Visualization in 2025

Ten packages, a decision tree, statistical plots and more

Jan 27 121

Vijay Gadhave

When to Use COUNT(*) vs COUNT(1) in SQL Queries

Note: If you're not a medium member, CLICK HERE

Jan 14 136 8

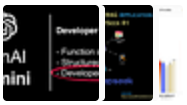
Lists

Practical Guides to Machine Learning

10 stories · 2185 saves

ChatGPT prompts

51 stories · 2533 saves

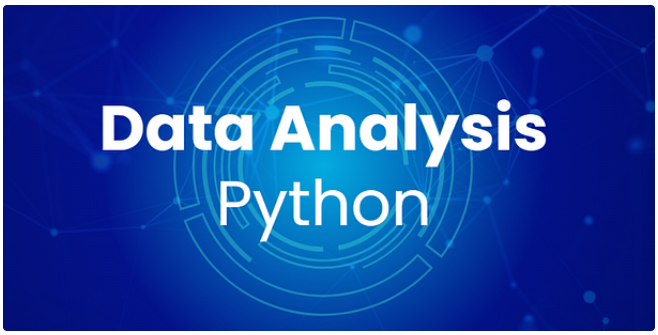


Natural Language Processing

1916 stories · 1571 saves

Staff picks

807 stories · 1603 saves



In Stackademic by Khouloud Haddad Amamou

Data Analysis with Python Pandas and Matplotlib (Advanced)

1. Introduction

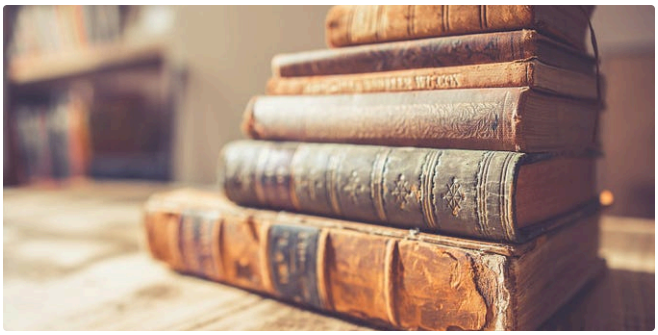
5d ago 95

In Towards AI by Egor Howell

Best Laptop For Data Science

What's the best laptop and does it even matter?

Jan 16 141 3

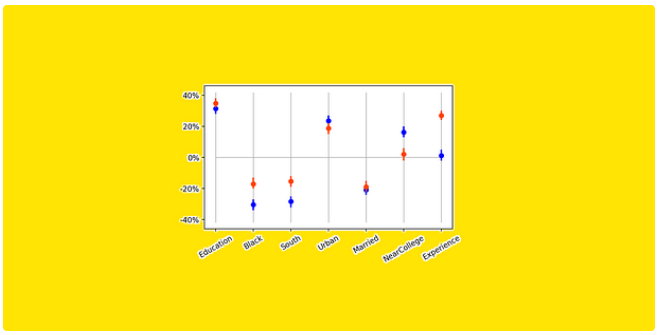


In Cogni.tiva by Axel Casas, PhD Candidate

5 Books That Will Make You Smarter Than 97% Of People

And embrace effective lifelong learning

Jan 12 1.8K 32



In Towards Data Science by Samuele Mazzanti

Think Correlation Isn't Causation? Meet Partial Correlation

Despite being so powerful, partial correlation is perhaps the most underrated tool in data...

Jan 8 961 18

See more recommendations